IDENTIFICATION OF APPLICATION BEHAVIOR USING PROCESS PROFILES

BY

ARNAV BHANDARI

A Thesis Submitted to the Graduate Faculty of

WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science

August 2019

Winston-Salem, North Carolina

Approved By:

Errin W. Fulp, Ph.D., Advisor

Todd Torgersen, Ph.D., Chair

William Turkett, Jr., Ph.D.

Acknowledgments

I would like to express my sincere gratitude towards my Advisor, Dr. Errin Fulp, for his guidance and patience throughout. This would not have been possible with his open door policy and unparalleled mentorship.

I would like to thank Dr. Todd Torgersen and Dr. William Turkett for their time and knowledge. They served on my committee and provided critical feedback that set me on track for this thesis.

I would also like to acknowledge my lab members, Katherine Juarez and Yixin Zhang who worked alongside me on this project.

Lastly, I would want to thank my family and friends for their love and support always.

Table of Contents

Acknowledgments ii

List of Figures vi

List of Tables vii

Abstract ix

Chapter 1 Introduction 1
- 1.1 Cloud computing Applications 1
- 1.2 Thesis Objective 2
- 1.3 Application Behavior 2
- 1.4 Previous Methods for Determining Behavior 3

Chapter 2 Application Behavior Identification 4
- 2.1 Applications and Behavior 4
- 2.2 Static Analysis 5
- 2.3 Dynamic Analysis 6
- 2.4 Limits of Static Analysis for Malware Detection 6
- 2.5 Static Analysis vs. Dynamic Analysis 7

Chapter 3 Data Processing and Creation of Features 9
- 3.1 Stages of Program Execution 9
  - 3.1.1 High Level Language 9
  - 3.1.2 Machine Code 10
  - 3.1.3 Assembly Language 10
- 3.2 Snooper Overview 11
- 3.3 Overview of the Intel Instruction Set 13
- 3.4 Linux Process Induced Instructions 14
- 3.5 Application Instructions versus Library Instructions 17

Chapter 4 Process Profiles and Mapping Schemes 19
- 4.1 Instruction Profiles 19
  - 4.1.1 Limitations of Instruction Profiles 20
- 4.2 Sequence Profiles 21

4.2.1 Spectrum Kernel 23
4.2.2 Limitation of Sequence Profiles 23
4.3 Mapping Schemes 24
4.3.1 Limitations of Mapping Schemes 26

Chapter 5 Clustering and Methodology 27
5.1 Supervised and Unsupervised Machine Learning 27
5.1.1 Supervised Learning 28
5.1.2 Unsupervised Learning 28
5.2 Classification 29
5.3 Clustering 30
5.4 k-Means Clustering 30
5.4.1 k-Means Algorithm 31
5.4.2 k-Means Complexity Analysis 32
5.4.3 Limitations of k-Means Clustering 33
5.5 Clustering vs. Classification 34
5.6 Silhouette Method 35

Chapter 6 Experiments 37
6.1 Process Usage Classes 37
6.2 Toolchain 38
6.3 The Effect of Command Line Arguments on Profiles 40
6.4 Impact of Library Instructions on Clustering 41
6.5 Clustering using Instruction Profiles 44
6.5.1 File Display vs. File System Interaction 45
6.5.2 Encryption vs. File System Interaction 46
6.5.3 File Display vs. Disk Management 47
6.5.4 Experimental Analysis 48
6.6 Clustering using Sequence Profiles 49
6.6.1 Experiments varying Mapping Schemes 50
6.6.2 Experiments varying Sequence Lengths 54

Chapter 7 Conclusions and Future Work 59
7.1 Future Work 61

Bibliography 62

Appendix A Instruction Profile 65

Appendix B Mapping Scheme Details 66

Appendix C Instruction Profiles Usage Class Experiments 67

Curriculum Vitae 68

List of Figures

3.1 Stages of a program's execution. 12

4.1 Instruction profile for an execution of the `ls` command. 20
4.2 Impact of mapping schemes to a new alphabet of size 28, 29, 4 and 3 on the total number of sequences for sequence profiles 25

5.1 k-Means as a step by step process [19] 32

6.1 Overview of the tools and toolchain used for experiments. 39
6.2 Confusion matrix for `head` and `tail` with and without library calls . 41
6.3 Silhouette scores for `ls` vs. `cat` with a sequence length of 4 with different mapping. 51
6.4 Silhouette scores for `ls` vs. `cat` with a mapping to 3 labels and a different sequence sizes. 56

List of Tables

3.1 Example Snooper output for the execution of the Linux command `ls`. . . . 13
3.2 Segment of the Snooper output for a C program returning the value `0xdeadbeef`. 15
3.3 Segment of the Snooper output for a C program that assigns a variable, calls a system function, and returns the value `0xdeadbeef`. 16
3.4 Number of instructions executed for the Unix command `ls` and different target directories, with and without library calls. 18

4.1 Instruction counts for different executions (varying the target directory) of the `ls` command 20
4.2 Relationship between sequence length and number of possible sequences. 24
4.3 Number of instructions that map to 4 and 3 labels. 26

6.1 Process usage classes, their descriptions, and examples use for experiments. 38
6.2 Mean and standard deviation of the total sequence frequencies for various Linux command given 20 unique arguments. 40
6.3 Accuracy of different pairs of Unix commands with and without processing library calls. 42
6.4 Clustering of File Display and File System Interaction groups with different number of targeted clusters. 45
6.5 Clustering of Encryption and File System Interaction groups with different number of targeted clusters. 46
6.6 Clustering of File Display and Disk Management groups with different number of targeted clusters. 47
6.7 Silhouette scores for `ls` vs. `cat` with sequences length of 4 and different mappings. Best scores are denoted with square brackets []. 50
6.8 Silhouette scores for `ls`, `find`, `sha256sum` and `md5sum` with a sequence length of 4. Best scores are denoted with square brackets []. 52
6.9 Silhouette Scores for different profiles of `ls` with a sequence length of 4. Best scores are denoted with square brackets []. 53
6.10 Silhouette scores of `ls` vs. `cat` with a mapping scheme of 3 (intel3.map) and different sequence lengths. Best scores are denoted with square brackets []. 55
6.11 Silhouette scores for `ls`, `find`, `sha256sum` and `md5sum` with a mapping scheme of 3 and different sequence lengths. Best scores are denoted with square brackets []. 57

6.12 Silhouette Scores for different profiles of `ls` with a mapping scheme of 3 with different sequence lengths. Best scores are denoted with square brackets []. 58

C.1 Clustering of File Display and Encryption groups with different number of targeted clusters. 67
C.2 Clustering of File System Interaction and Disk Management groups with different number of targeted clusters. 67
C.3 Clustering of Encryption and Disk Management groups with different number of targeted clusters. 67

Abstract

A computer application will often process thousands of machine level instructions during its normal execution. This thesis seeks to develop a dynamic analysis of these executed instructions in order to infer the application's behavior or general notion of the task being performed. Knowing the behavior of an application can help improve system management by allowing more efficient allocation resources (e.g. processor time, memory, etc...).

In this thesis application behavior is inferred using a clustering technique, specifically k-means, where similarly behaving applications are expected to cluster together. These clusters are formed based on process profiles that contain information regarding the frequency of the machine instructions or the frequency of machine instruction sequences (an ordered list) used during execution. Instruction mapping techniques at the assembly level that consider both similarly functioning instructions as part of the same category and various length instruction sequences are also investigated. The experimental results of inferring the behavior of various Linux commands (utilities) indicate this proposed approach has considerable promise.

Chapter 1: Introduction

The proper management of computing resources, from smartphones to cloud services, can benefit from knowledge about the nature (behavior or characteristics) of executing processes. For example, in a cloud computing environment, processes that are computationally intensive (computational nature) may generally perform better on servers with faster processors. Even smaller computing platforms can benefit from such knowledge. Specifically, consider managing processes for a smartphone. Processes that require significant network activity or any calls to the Internet, including streaming, might be scheduled less frequently if network connectivity is limited. Enforcing security can similarly benefit from behavior identification, such as monitoring processes that require significant amounts of network access or disk activity.

1.1 Cloud computing Applications

Cloud computing is increasingly recognized as a more efficient approach for providing computing resources, such as storage and computing power. Using this model, resources are available on demand without direct management by the user. However, as the use of cloud computing and similar on-demand computing solutions increases, managing the actual shared computing infrastructure is becoming increasingly difficult. The types of applications encountered by cloud administrators can be quite diverse, ranging from network services to computational-oriented tasks. More specifically, these application may need or benefit from certain resources or system configurations (environments) that may not be known by system administrators, or even by the user. The benefits of more sophisticated management can result in power savings and is the focus of green computing endeavors [1].

Better resource management can be achieved if the ***nature*** of processes is understood. For example, network-oriented processes can be placed on systems that have higher bandwidth network connections, while computing-oriented process can be placed on systems that have higher clock rates and/or more processor cores.

1.2 Thesis Objective

The goal of this master's thesis is to identify certain specific behaviors that a process is showing considering what is expected from the process. This thesis investigates process behavior through various usage classes, and the similarities and dissimilarities across them are studied. There are other aspects that are also explored including inter-usage class and intra-usage class clustering. The objective is to have broad labels to be identified based on clusters formed using machine learning tools such as k-means. Labels will include, but not be limited to: process-oriented, file-oriented, network-oriented, and computational-oriented processes.The value of k used for k-means has been shown to be optimized using the silhouette method. These results can have a large impact on the way processes are viewed and provides a behind the scenes look at the nature of different UNIX utilities. This work is a critical step for better resource management and could be revolutionary in the way processes are managed.

1.3 Application Behavior

The behavior of an application depends on the context in which it was executed or the current state of the program. Therefore, behavior is best identified based on the application's actual execution. To account for this, a dynamic analysis of the application is ideal and is the method being proposed in this study. This would lead us to the true nature of the application and could significantly improve the way computing resources are managed to promote efficiency.

On a broader scale, let us consider the workings of a web browser. A browser itself can do very different things ranging from playing multimedia, browsing the Internet, to running extensive computational programs. Browsers also have the capability to run full- fledged compilers which can involve hundreds of thousands of lines of code generated and numerous library calls to run something trivial in a higher language. The browser fetches data from the web and also may use encryption and decryption. These examples show the many different types of behavior that a web browser is capable of. For this thesis what is most relevant is identifying the behavior of the browser at a given time, with a more indicative label of what is happening.

1.4 Previous Methods for Determining Behavior

Behavior is based on the actual execution of the program. The current method used for application identification is a static analysis. Since code is not executed in this analysis and the actual execution path is not considered, this approach is considered to be probabilistic. Therefore this would not be a good indicator of the actual behavior of the application being studied. Furthermore, a static analysis might not be possible in some cases when the source code is not available and only a binary is provided. This can happen in the case of system administrators where only the binary is provided for analysis, and a decision needs to be made as to what cluster/software would be ideal for running it.

Therefore a dynamic approach which involves the actual execution of the application is favored. Profiles are constructed based on either the counts or a sequence analysis of the assembly level instructions. This approach can be used to identify clusters and thereby identify behaviors similar to other applications and their behaviors. It is significantly harder to identify the behavior of an application without actually executing it or looking at a partial execution.

Chapter 2: Application Behavior Identification

One of the more important caveats of this research is understanding the difference between the application and the behavior of an application. Many different usage classes exist and represent the behavior of the different applications. For example, the Linux commands that are explored within this thesis belong to compression, networking, encryption, and other classes that will be explored in more detail later. The primary objective is to determine how similar various commands are within each of these groups to commands outside of these groups and how their behavior relates with one another. This study shows that application and behavior do not necessarily have to be the same thing. An application can have a range of different behaviors as illustrated in Section 2.1 The study can be done through either a static analysis or dynamic analysis but the specifications of this thesis favor the latter.

2.1 Applications and Behavior

The Linux command `tar` has various different *behaviors* based on the types of flags used [2]. First, the command `tar -xvzf tarredFile.tar.gz` would extract all the files from the zipped file and also print the file names as they are being extracted. Second, the `tar` command to list and search for the contents of a tar archive would be: `tar -tz -f tarredFile.tar.gz.` Third, the command: `tar -cvzf tarredFile.tar.gz ./PathtoFiles` would compress all the files within the directory `PathtoFiles` and create the gzipped file called `tarredFile.tar.gz`. [2]

There are multiple other flags that would make `tar` perform in other capacities but just from looking at these three different cases, the variety of different behaviors

this single Linux command has becomes apparent. Taking a slightly closer look at the three cases, the behavior of the first case closely resembles that of a decompression and even an output, possibly to a terminal. The second behavior would more closely resemble a file output and would be similar to the Linux command `ls` which displays all the files in the current directory. The third case would resemble behaviour closest to other compression commands.

This one command can have at least these three separate behaviors, and this behavior is what we are interested in identifying. What this means is that the application that is being tested is *tar*, while the behavior is actually across three separate categories which will be identified as usage classes. A behavior is not restricted to one usage class. This behavior recognition and identification is of critical importance to this study.

2.2 Static Analysis

Static analysis involves the study of code without its actual execution. This method reasons over all the possible outcomes that could be expected at runtime by looking at the source code [3]. It also explores how often and which functions are called, performs dead code analysis, and mathematically proves various properties of the code under analysis. This method does not take into consideration the values input into the function and simply analyzes the function as a whole in the hope of identifying the behavior. This method is conservative and uses an abstract model to predict behavior.

The disadvantage of this method is that this level of abstraction could provide a probabilistic behavior of code. A lot of extra information is preserved and would therefore be analyzed as part of the static analysis without considering the actual execution branch, thereby skewing the relevance to the actual behavior of the code.

For example, if a program has multiple if and else statements but only one case is relevant based on the input to the program, a static analysis would not be able to identify and emphasize the relevant case. On the other hand the advantage of this method is that all information is preserved regardless of its potential [3].

2.3 Dynamic Analysis

This method involves the actual execution of the source code. An advantages of this method is that there is no ambiguity since an execution branch has been selected. Therefore this is not a probabilistic study like static analysis [3]. There is no abstraction needed for this method, and the results are definitive. Some of the disadvantages of this method are that compiling and then running a simple C++ program requires the analysis of thousands of lines of assembly language instructions. The other issue is reproducibility since there is no guarantee that the results that have been obtained for an earlier run can be extrapolated for future runs [3]. Finally there could also be runtime errors or faulty inputs to functions which could skew the results of a dynamic analysis.

2.4 Limits of Static Analysis for Malware Detection

A traditional method for identification of malware is through static pattern matching techniques such as virus and spyware scanners. However, this approach is easy enough to confound by utilizing non complex code transformations. This leads to the development of more powerful scanners which identify malware using semantic signatures. This approach includes techniques such as model checking and automated theorem proving to perform malware detection. This proved to be effective against an adversary which was unaware that these techniques were being used. [4]

To further confound static analysis methods, a technique known as binary program

obfuscation using opaque constants was developed to obscure the control flow of programs and mask the ways in which different variables and constants are accessed. This method is based on the known NP-complete problem `3SAT` [4]. As soon as this complexity class is breached there is no computationally feasible solution to these answers. We could get good solutions using other heuristics and clever methods, but there is a limit to all of this when we get into this NP-complete territory. Interestingly enough, dynamic analysis is not at all vulnerable to any of these techniques, while static analysis has a limit on how well it can perform [4]. Due to this limit on the performance of static analysis and no such limit for dynamic analysis, the latter could be considered to be a better approach.

2.5 Static Analysis vs. Dynamic Analysis

One of the biggest motivations to use dynamic analysis is that all the information and the execution path are not available until runtime. Static analysis can take place before the actual execution of the program but overcompensates in abstraction compared to what the dynamic analysis does. The other advantage of a dynamic analysis over static analysis is that compiler optimization techniques do not need to be considered. This includes but is not limited to dead code elimination and constant folding.

Conversely, the fact that static analysis lets us verify all possible execution paths can have tremendous value. However the scope of this project is limited to just one possible path which is the execution path and would best indicate the behavior of an application. Therefore static analysis did not seem like the best choice for computationally evaluating the behavior of the processes. A dynamic analysis involves executing each process to determine what code is generated, and then this data is transformed into profiles for further analysis. Static analysis for a single process

would examine solely the specific code regions that the command accesses. This is independent of many of the variables one would typically see during runtime, for example the environment and even flags which the process would run with. These are variables which can significantly impact the profiles that are generated which would impact the way clustering occurs and finally would skew what is perceived to be behavior [3].

Chapter 3: Data Processing and Creation of Features

The objective of this thesis is to identify process features that will help indicate the behavior of a process. In computer science, a process refers to an application executing on a processor. This chapter details how and when process data is extracted during execution. This is primarily done through a process control program called *Snooper*, which described in this chapter.

3.1 Stages of Program Execution

There are multiple stages that the process execution chain goes through, converting data from one form to another. The procedure consists of a single process from the moment the process begins execution to its termination. It has multiple levels which are explained in the following subsections. A program is written in a high level language which is not comprehensible to the machine. This needs to go through levels of change and compilation before it can be actually run on the hardware of the machine in the form of machine code. This machine code then interfaces with the hardware to run the program.

3.1.1 High Level Language

High level programming languages for example languages like C/C++, MATLAB, and Java are easier for programmers to understand (and develop in) than lower level languages such as MIPS and Intel x86. Higher level languages provide a certain level of abstraction which allows for a computer independent development. This is because these languages are converted from a higher level to a form of machine readable code

which executes on the hardware. As a result of this abstraction, a single instruction in a high level language translates to multiple instructions at the machine level. This is done through either the use of an interpreter or a compiler. High level languages are considered human readable unlike machine code.

3.1.2 Machine Code

This is the lowest level of software instructions closest to the hardware and consists of only 0s and 1s. These represent the on or off states based on electrical impulses. A set of 0s and 1s is called an instruction, and different hardware have different instruction lengths that they process. Each instruction consists of two parts, an opcode and operand(s). Every opcode has a corresponding instruction, and then it uses the elements in the operands to perform that specific instruction. The operands can be data values, memory addresses, or even registers [6].

3.1.3 Assembly Language

Looking at the different levels of abstraction that exists, assembly language lies in between a high level language and machine code. Sometimes during the translation cycle the high level code is converted to assembly level code which is then converted into machine code to interface with the hardware. In some cases, higher level languages are converted directly into lower level machine code. This type of code is considered to be low-level as it is still readable by a human user, but lots of abstraction has been removed. Here, a memory to memory transfer or a direct access to the registers which is not possible at the higher levels can take place. Each line of assembly language can be broken up into four separate parts. These include the label, mnemonic, operands, and the comment. The mnemonic is also referred to as the opcode which directly translates to the opcode of a machine level instruction. An

example of an x86 assembly language instruction is given below [7]:

```
LOOP: MOVB r0, #80 ; initialize counter
```

The first part of this example, `LOOP`, is a *label*. A label is associated with the address of the instruction for the code segment. Any reference to `LOOP` after it has been defined will correspond to this memory address. `MOVB` is the opcode which is a human readable indicative of what the instruction actually does. `MOVB` will take in 2 operands and store the value of the second operand into the first operand. `r0` and `#80` are the two operands in this example. There do not have to be 2, but in this case the hexadecimal value of 0x80 is being stored into register 0. The last part of the assembly language instruction, denoted by a semicolon, is optional and is the comment. The corresponding machine level instruction for this line of assembly language code is `11 0000 1000 0000`. Every line of assembly language will have a corresponding machine code instruction [7].

3.2 Snooper Overview

Since this thesis is working with dynamic analysis, a tool needs to be used which can capture code of the running program. Ideally when looking at the stages of program execution, accessing information closer to the machine level would be more indicative of what is being executed. Considering the fact that this research is expected to explore the behavior of the application, a static level analysis would not work. As mentioned in Section 2.2, the static analysis provides a broad overview of the higher level language with probabilistic models, but the specifics of the lower levels still have to be inspected. This means that the point of data interception should be at a lower stage of the program execution. Ideally the data that is considered at would be the

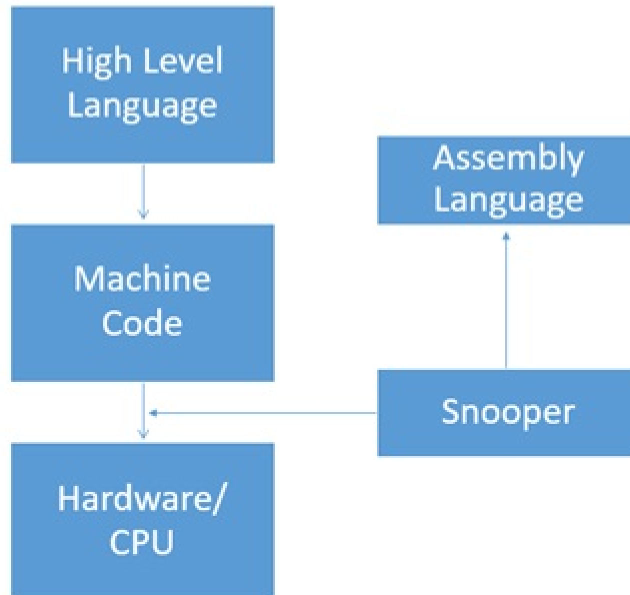

Figure 3.1: Stages of a program's execution.

machine level instructions since that is what is actually being processed.

Snooper is a process control program that can intercept the executed machine instructions of another running process. As Figure 3.1 shows, Snooper intercepts execution at the machine code level and provides an output in a assembly language for further processing [8]. As a result of observing the actual instructions executed, there is no ambiguity or any of the issues associated with static analysis.

The original Snooper program was made by Arian Stolwijk [8] and then was edited further by the Wake Forest team before it was adapted for this research project. The Snooper program dynamically disassembles x86 instructions. This program makes use of `ptrace` and currently only works on single threaded Linux utilities. It forks, and the child process runs the application under investigation while the parent process saves the register contents after each instruction of the child is executed. By this method, Snooper is able to record the instructions executed by the application. Snooper can be run on any application and can conduct this disassembly process on any executable.

Address	Machine Code	Assembly
⋮	⋮	⋮
0000000000412514	31db	xor ebx, ebx
0000000000412516	48c1fd03	sar rbp, 0x3
000000000041251a	4883ec08	sub rsp, 0x8
000000000041251e	e845fcfeff	call 0xfffffffffffefc4a
⋮	⋮	⋮

Table 3.1: Example Snooper output for the execution of the Linux command `ls`.

Example Snooper output is given in Table 3.1. In this example Snoper was used to record the executed instructions for the Linux command `ls`.

This shows the usefulness of Snooper since it is able to obtain interpretable output in assembly language which programmers can understand. Since Snooper is acting on such a low level of the process execution path, this gives a stronger indication of what is truly happening behind the scenes and which instructions are being called, strengthening this dynamic analysis. This set of instructions is part of the x86 instruction set which is detailed in Section 3.3

3.3 Overview of the Intel Instruction Set

On June 8th, 1987, Intel launched the 8086 which was the first 16- bit microprocessor. The set of instructions executed by this family of processors is generally referred to as the x86 instruction set. Over time this set of instructions has grown as new processors have been introduced. This processor is now one of the most widely used processors, and research in this sphere could be extremely beneficial. [12]

This research is conducted on an Intel(R) Xeon(R) CPU E5-2699 v3 2.30GHz which is a Complex Instruction Set Computing (CISC) type of architecture. This type of microprocessor was designed to execute multiple instructions within a single execution cycle. For example, when we are looking at an add instruction and are

adding the values of two registers to one another, then we are retroactively also perform load instructions to get to that data. The x86 instruction set is compatible with CISC.

The x86 instruction set consists of instructions that are similar to assembly language instructions which can perform certain actions. There are over 1,000 different valid x86 instructions. However, in this thesis, only the set of 723 valid instructions that are produced experimentally is studied.

3.4 Linux Process Induced Instructions

When a process executes within a modern operating system, several tasks must occur to ensure the proper hand-off from the operating system to the process. This set of starting tasks can be referred to as the *wind-up*. Similarly, hand-off also occurs back to the operating system when the process is complete, and this is called wind-down. Both wind-up and wind-down are observable from Snooper; however, this portion of execution may not be useful for behavior identification since it could be a constant in each profile created.

Experimentally, it was discovered that the addresses associated with the application had the form `0x0000000000400xyz`. The `xyz` at the end is a variable that depends on the compiler or the application. The other crucial bit of information that is observed is that non-application instructions seem to have an address in the form of `0x00007f324buvwxyz`. The `uvwxyz` at the end is a variable which changes based on the specific library that is being referenced at that specific time. This is the standard by which Linux organizes memory for processes. The start-up and wind-down code are very similar as shown by the next two examples. This can be illustrated through the following simple C program:

```c
int main()
{
    return 0xdeadbeef;
}
```

Snooper was run on this trivial C program and resulted in the execution of 92,303 instructions. The other crucial information here within the Snooper output is that the main actually only started on instruction number 90,693.

In essence, the main program is simply saving the `rbp` which is the previous stack frame, moving the value of `0xdeadbeef` into the `eax` register, and then leaving the main function, and then the `ret` signifies the end of the function. Table 3.2 shows the main program on Lines 90695-90699. This shows the partition in the application code from the Linux code library calls by observing the difference in the hexadecimal and address representation. The code that occurs before the main actually executes can be associated with the work that is needed to setup the program to actually execute.

Lines	Hexadecimal and Address Representation	Snooper output
⋮	⋮	⋮
90693	00007ff6bd7ebb3e 488b442418	mov rax, [rsp+0x18]
90694	00007ff6bd7ebb43 ffd0	call rax
90695	00000000004004b6 55	push rbp # enters main function
90696	00000000004004b7 4889e5	mov rbp, rsp # save the stack pointer
90697	00000000004004ba b8efbeadde	mov eax, 0xdeadbeef # stores hex
90698	00000000004004bf 5d	pop rbp # leaves the main function
90699	00000000004004c0 c3	ret # end of the function
90700	00007ff6bd7ebb45 89c7	mov edi, eax
90701	00007ff6bd7ebb47 e814600100	call 0x16019
⋮	⋮	⋮

Table 3.2: Segment of the Snooper output for a C program returning the value `0xdeadbeef`.

After looking at this result, a slightly more complicated problem was looked at to see how the output would vary. In the next example, a system call has been introduced into the C program, and the effect of this on the snooper output is observed. The C

program is defined as follows:

```c
int main()
{
    int var = 0xdeadbeef;
    write ( stdout , &var , sizeof(var));
    return 0xdeadbeef;
}
```

This is an important avenue to study since programs of any complexity would rely on these types of system calls. Therefore studying the effects of such calls on the Snooper output would be beneficial to the research. The Snooper output obtained is 93,601 which is 1,298 more instructions than just the return of `0xdeadbeef` which was the previous example. Segments of the output are given in Table 3.3.

Lines	Hexadecimal and Address Representation	Snooper output
⋮	⋮	⋮
91305	00007fc7b78a5b3e 488b442418	mov rax, [rsp+0x18]
91306	00007fc7b78a5b43 ffd0	call rax
91307	0000000000400546 55	push rbp
91308	0000000000400547 4889e5	mov rbp, rsp
91309	000000000040054a 4883ec10	sub rsp, 0x10
91310	000000000040054e c745fcefbeadde	mov dword [rbp-0x4], 0xdeadbeef
91311	0000000000400555 488b0504042000	mov rax, [rip+0x200404]
91312	000000000040055c 488d4dfc	lea rcx, [rbp-0x4]
⋮	⋮	⋮
91322	0000000000400416 ff2514052000	jmp qword [rip+0x200514]
91323	00007fc7b7c434e0 4883ec38	sub rsp, 0x38
⋮	⋮	⋮
91994	00007fc7b795fc49 c3	ret
91995	0000000000400575 b8efbeadde	mov eax, 0xdeadbeef
91996	000000000040057a c9	leave
91997	000000000040057b c3	ret
91998	00007fc7b78a5b45 89c7	mov edi, eax
⋮	⋮	⋮

Table 3.3: Segment of the Snooper output for a C program that assigns a variable, calls a system function, and returns the value `0xdeadbeef`.

The main point is that `0xdeadbeef` appears in 2 places in the entire snooper output. This is in line 91,310 and line 91,995. Lines 91,307-91,322 contain application code within which the first instance of `0xdeadbeef` is. The next time `0xdeadbeef` is found is in the consecutive block of application code is on line 91,995 which is 685 instructions later. The conclusion that one can draw from this is that there are numerous calls to various libraries in between calls to the main program where `0xdeadbeef` is stored as a variable and then when it is returned `0xdeadbeef`. This indicates that there are application instructions interleaved with the library instructions.

Just the addition of 2 more lines of code in the C program led to more than a 1,000 more lines of assembly level instructions that were needed to be processed. This is a very trivial increment in a higher level language but has a major impact when instructions are considered at a lower level.

When the output was considered for both of these examples, the other crucial information that is important for this research project is that the start-up and the wind-down code segments look very similar. The only substantial difference appears to be that there is a difference in the memory locations of where things are stored but it still follows the same general format one would expect from library call and application instructions.

3.5 Application Instructions versus Library Instructions

The previous section provides examples of how Snooper reacts to different C programs and how library calls impact the executed instructions. Now as an example consider the execution of `ls`, a common Unix utility/application. The `ls` command is designed to provide information about files on the computer system. The following will demonstrate how the executed instructions for `ls` will vary based on what file (or directory) is the intended object of the command. These `ls` commands are different

Command target (directory)	No. of app instructions	Total instructions	Percentage of application instructions
`/dev/block`	1438	526437	0.273%
`/dev/bsg`	1376	526363	0.261%
`/dev/bus`	1376	526363	0.261%
`/dev/char`	1407	526400	0.267%
`/dev/cpu`	1376	526413	0.261%

Table 3.4: Number of instructions executed for the Unix command `ls` and different target directories, with and without library calls.

from each other on the basis of the directory that they are being asked to display.

Table 3.4 outlines 5 specific `ls` execution examples and their counts (number of instructions executed). The chosen directories are diverse in the counts of files and folders/sub folders. From this it can be extrapolated that the percentage of instructions that are specific to the application (non-library) are around 0.265%. If a small percentage of instructions can be examined and it can still be determined to be an `ls` command then this indicates that library calls are not actually adding any value for process behavior identification.

Chapter 4: Process Profiles and Mapping Schemes

Inferring the behavior of a process will require a representation of the instructions performed during execution. This could simply be the actual list of executed instructions; however, even a simple application (e.g. a Unix command) can execute hundreds of thousands of instructions, as shown in the previous chapter. Therefore relying on instruction lists would not scale for larger applications.

For this thesis, process behavior will be inferred using a *process profile* which is a condensed representation of the executed instructions. Process profiles can be either an *instruction profile* or a *sequence profile*, where each provides a different representation of the instructions executed.

4.1 Instruction Profiles

Given the instructions executed by a process, an instruction profile is the relative frequencies of each unique instruction. A graphical representation of an instruction profile for the execution of the Unix command `ls` is given in Figure 4.1. From this example, it can be observed that the most executed instruction is `mov` with a count of more than 400.

Table 4.1 also provides us with the difference in unique frequency for each profile with library calls included and excluded. The difference is considerate since the drop is from approximately 92 to 37 unique instructions when library calls are removed. This would be the length of each of the profiles, and each instruction would have a corresponding frequency.

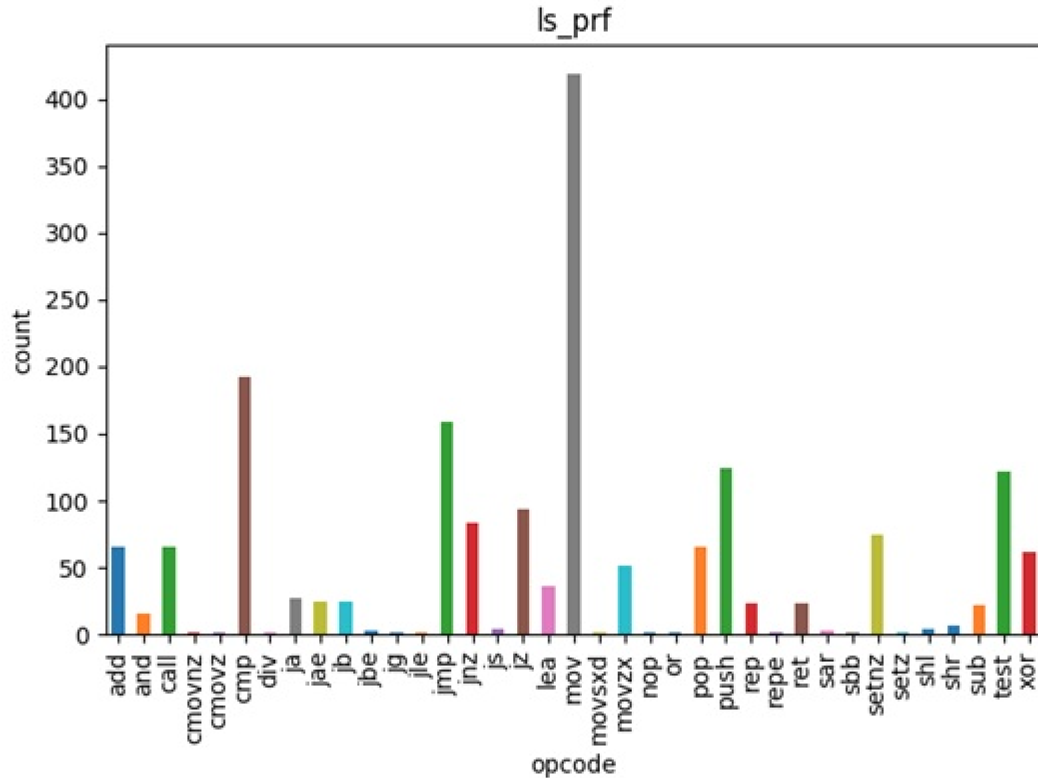

Figure 4.1: Instruction profile for an execution of the `ls` command.

Directory	No. of application Instructions	No. of unique Instructions	No. of unique Instructions without Library calls
/dev/block	1438	92	37
/dev/bsg	1376	92	37
/dev/bus	1376	92	37
/dev/char	1407	93	37
/dev/cpu	1376	92	37

Table 4.1: Instruction counts for different executions (varying the target directory) of the `ls` command

4.1.1 Limitations of Instruction Profiles

Although simple, instruction profiles provide a good representation of executed instructions. Instruction profiles also provide commonality between different processes

for comparison purposes. This method has merit, but there is also much data that is being lost in this process. When a relative frequency is considered for each unique instruction, valuable information is lost since instruction ordering is not preserved. If there is a certain sequence of instructions that is unique to only a certain application, then this information would not be able to be recovered on the basis of instruction profiles.

Another potential issue for instruction profiles is related to the x86 instruction set, specifically the CISC architecture. Within the x86 instruction set there are several different instructions that essentially perform the same task; however, these instructions would be counted separately because of their semantics. For example considering the instructions in Appendix A, there are 2 `cmovnz`, 2 `cmovz`, 371 `mov`, 2 `movsxd` and 27 `movzx` which essentially are all perform the same task. These instructions move values from address to other addresses or from register to register or even from address to register or the other way around. The variations with some of these `mov` instructions could also be because of the compiler or any specific flags that were included at compile time. A possible approach for treating functionally equivalent instructions the same is to associate (map) these instructions to one common label. So for example, all move instructions become simple `mov`.

4.2 Sequence Profiles

Although instruction profiles provide a condensed representation of the instructions executed by a process, there is no information provided about the order of the executed instructions. To maintain the ordering of instructions, another type of profile can be made which is the sequence profile. This would put more emphasis on the sequence of instructions rather than how frequently an instruction occurs. Each sequence has a length which stays constant through an experiment. A sequence is defined as a set

of consecutive executed instructions of specified length found in the Snooper output; therefore, a sequence profile would be the frequency of each sequence.

A sequence is a fixed-length, ordered list of instructions that are executed, and a sequence profile is a list of counts of every possible sequence observed during execution. There are two important characteristics of a sequence: the length of the sequence (which is fixed) and number of possible instructions, referred to as the alphabet size, that form the sequence. The process is best explained using an example. Consider finding the instruction sequences of length 3 given a list of 6 instructions that are executed in the order they appear.

```
1 mov rax, rbx
2 call rax
3 mov rax, rdx
4 call rbx
5 mov rdx, rbx
6 push rbp
```

Processing the instructions operates similarly to a sliding window, where the window size is the same as the length of the sequence. For the example, the window initially contains the first three instructions `[mov, call, mov]`. Note, only the opcode is of interest, not the operands. These three instructions form a sequence, and a counter is initialized to one, indicating that this sequence was executed. After the initial window is processed, the window slides-down by one instruction and now contains `[call, mov, call]`. The counter representing this sequence is incremented, and in this example, the counter value would be 1 since it is the first observance. The window slides down one instruction and now contains `[mov, call, mov]`. The counter associated with this sequence is incremented and becomes 2 since this is the second time it has been observed in a window. This process would repeat until the last 3 instructions are processed.

4.2.1 Spectrum Kernel

By design the spectrum kernel is a straightforward method and does not depend on any generative models. This method was constructed by Leslie [9] for determining the similarity between protein sequences. The spectrum represents the frequency values of every possible s-length sequence bounded by the alphabet. To find all the sequences that are present in the kernel, there is a sliding window of sequence length s that is passed over the data (in this project, the list executed machine instructions), and the sequences are recorded [9] [10].

It is very important to preserve every single potential sequence that is encountered for a sequence profile to be effective. In this regard, the sliding window is extremely useful in preserving this information.

4.2.2 Limitation of Sequence Profiles

An important attribute of sequence processing is the number of possible sequences that can exist. Consider the possible sequences of length s given a set of unique items. Let the variable a represent the number of possible items, which is often referred to as the alphabet size. For this thesis, the alphabet would be the set of x86 instructions. The relationship between the sequence length s, alphabet size a, and the number of possible sequences n is given by the following equation,

$$n = a^s \tag{4.1}$$

Therefore, even a small increase in sequence size would lead to orders of magnitude of an increase in the number of sequences. There are a total of 723 unique x86 machine instructions that are considered for this research. Inferring process behavior becomes computationally difficult, in terms of machine learning and clustering component, if long sequences are required. This is shown in Table 4.2, where a sequence length of

16 instructions can result in over 5×10^{45} possible sequences. If longer sequences are desired, it may be possible to reduce the number of possible sequences by decreasing the alphabet size. For this thesis, alphabet reduction is possible using a mapping scheme where multiple instructions that do essentially the same task are represented with one unique label.

Sequence Length (s)	Number of Possible Unique Sequences
1	723
2	522729
4	2.73246×10^{11}
8	7.46632×10^{22}
16	5.57459×10^{45}

Table 4.2: Relationship between sequence length and number of possible sequences.

4.3 Mapping Schemes

There are a lot of instructions that are doing similar operations; therefore if a label can encapsulate all these similar instructions, then the alphabet can be significantly reduced. This reductions would significantly impact sequence profiles since the relationship between the number of possible sequences and the alphabet is exponential, as shown in Equation 4.1. This becomes apparent in Figure 4.2 which shows the impact on the number of sequences when the alphabet is reduced for sequence profiles. A mapping scheme allows for a smaller number of total sequences generated when profiles with large sequence lengths are considered by reducing the alphabet.

Mapping is not as useful in reducing instruction profiles when compared with sequence profiles. However this technique can still be applied to reduce the total amount of features that are generated. For example, Appendix A contains the instruction profile of a `ls` command and has 2 `cmovnz`, 2 `cmovz`, 371 `mov`, 2 `movsxd` and 27 `movzx` which can be broadly categorized as `mov` commands. This would effectively

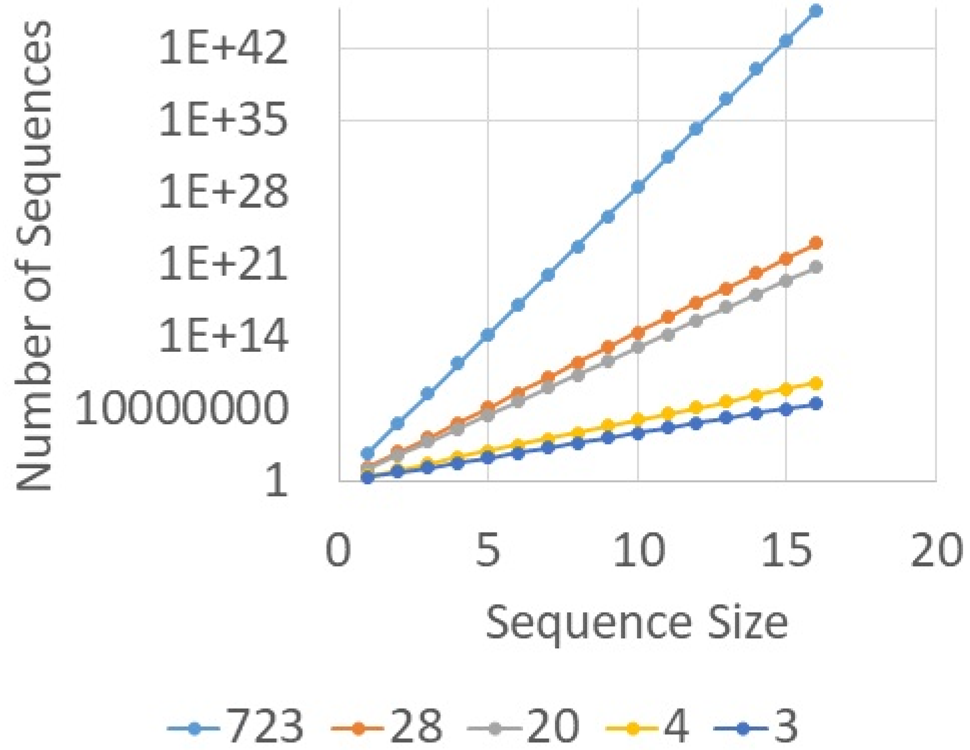

Figure 4.2: Impact of mapping schemes to a new alphabet of size 28, 29, 4 and 3 on the total number of sequences for sequence profiles

map 5 separate instructions to 1 label.

Experimentally, there are a total of 723 different x86 instructions that are seen, and mapping schemes have been created to reduce 723 to 28, 20, 4 and 3 labels. The scheme to map to 28 and 20 were taken from documentation [11]. The scheme to map to 4 categories and 3 categories are shown in table 4.3. Intel map of 28 and Intel map of 20 are in Appendix B.

Intel map 4		Intel map 3	
No. of ins.	Label	No. of ins.	Label
207	ARITH	207	ARITH
68	COND	224	CONTROL
156	CONTROL	292	MOVE
292	MOVE		

Table 4.3: Number of instructions that map to 4 and 3 labels.

The labels that were chosen are distinct, and table 4.3 provides details on the number of instructions which map to each label. These labels are broadly indicative of what is happening at the assembly level. If there are a lot of arithmetic based operations, the process might be more calculation based. If there are a lot of move operations, then there could be a lot of movement of data for potential display purposes or other forms of system interaction. These labels could be indicative of behavior and are extremely effective in reducing sequence profiles.

4.3.1 Limitations of Mapping Schemes

Although smaller alphabet sizes allow for larger sequence sizes, there is a loss of instruction specificity which may be important when distinguishing process behavior. For example, if all the x86 instructions were mapped to just 3 broad labels there would be a loss in specificity, but the sequence profile would have a far smaller feature space.

The mapping scheme that has been proposed could be developed further. This scheme has been taken from documentation [11], but there could be better mapping schemes. One of the biggest uncertainties with mapping is that the mapping scheme is man-made and therefore prone to bias. This means that the best choice could be determined through experimental values and their results. There should be an optimal region where the number of labels are not too big, so that the alphabet is sufficiently reduced, and not too small where there is a loss of valuable data because of overgeneralization.

Chapter 5: Clustering and Methodology

One approach for determining if process profiles can be associated with process behavior is to cluster the profiles of various application executions. As a result, the instruction and/or sequence counts, found in the profiles, become the features used to represent a process. The hope is that only similarly behaving processes will cluster together, thus indicating that profile contents (instruction and/or sequence counts) can suggest behavior. This research uses machine learning tools like k-means clustering to identify how similar processes are grouped together based on instructions sequences. If similar behaving processes cluster together, it may indicate that it is possible to infer behavior based on process profiles. This chapter will describe supervised and unsupervised learning and will explain why clustering was decided upon over classification techniques. The other big part of this project is the silhouette method for identifying the ideal number of clusters, and this method is introduced in this chapter.

5.1 Supervised and Unsupervised Machine Learning

Broadly speaking, machine learning can be broken up into three categories which are supervised, unsupervised, and reinforcement learning. There are also variations of machine learning which include elements of these three categories, for example semi-supervised learning. However for the scope of this project, the most relevant machine learning techniques are supervised and unsupervised learning, and a brief explanation is given for each of these methods in this chapter.

Some background on the machine learning process is important to know. A feature is defined as the set of criteria based on which decisions are going to be made. Each

data member can have multiple features which is indicative of what the input to the machine learning algorithm will be. A label closely ties with what the resultant output should look like. The datasets are usually broken up into training sets and testing sets. The training set is what the machine learning algorithm learns from, and the testing set is what the algorithm is tested on to calculate the performance of the algorithm.

5.1.1 Supervised Learning

For this type of learning, the training set has to have their corresponding labels. What this means is that the training data is encoded in pairs. The dataset can be defined as a vector of input values. Every single vector of input value i has a corresponding output vector value o. The machine learning function that is generated is a function F where $F(i) = o$. This function is dependent on a set of parameters, and the goal is to adjust these parameters to maximize the correct number of input i which would successfully map to their corresponding output o [13].

There are various types of features used to represent the input, and the transformation function can have varying degrees of complexity depending on the chosen model. One of the big limitations of supervised learning is that one could overfit for the training data, and if an example occurs which the algorithm has not seen before it could perform very poorly. Ideally the best model would have a certain level of abstraction so that underlying patterns can be detected [13].

5.1.2 Unsupervised Learning

In this type of machine learning, the type of data that is available to us is only the input data i. There is no correct answer that is available to us, and here the learning is expected to identify underlying patterns. This type of learning is focused

on the correlations between features and if clustering can successfully occur based on similarities between data points. The other important component of this type of learning is whether outliers can be detected and how that impacts the results [13].

Some of the uses of unsupervised learning are finding clusters, reducing the dimensionality of a problem, finding hidden cases and outliers in the data, and even modeling the density of the data. In this case, the correct answer may not be known and the analysis could be dependent on the variety of input data [14].

5.2 Classification

Classification is a type of supervised machine learning which involves taking the input that needs to be classified and assigning it potential pre-defined class labels which would be the output of the classification method. In this type of learning the data that contains the actual output values is available. Broadly, these are the two big steps for classification.

The first step for the classification process is the construction of a mapping scheme or a function. Let us assume that the prediction label $y = f(X)$ where X is a tuple of given input data. The mapping function would ideally be able to create a distinction between the data classes. This function represents a set of classification rules, decision trees, or mathematical formulae. These rules will train on a segment of the data and then can be used to categorize future data tuples [15].

The second step of the classification process is the actual testing of the model that has been constructed. If the testing takes place on the same set of data as the training, then the accuracy could be skewed due to overfitting. Overfitting is when there is no general trend that has been identified, and only the data that has been trained on would get an accurate result. A test set independent of the training set would be used, and the accuracy would be calculated. This would be the percentage

of test tuples that are classified correctly [15].

5.3 Clustering

Clustering is also referred to as cluster analysis and is a type of unsupervised machine learning where the entire set of data is partitioned into subsets. This partitioning is done in such a way that the level of association between two data points is maximum when they belong to the same subgroup and minimum otherwise. This type of analysis is used to discover structures and patterns within the data without any background explanation or bias. This process discovers structures in data on the basis of proximity without going into the details of why they exist [16].

Clustering has many applications and can be a very powerful machine learning algorithm. It is useful in image segmentation, exploratory pattern analysis, decision making and data mining. In these cases there is little background knowledge about the datasets that are being examined. There might be some statistical models which would give an idea of what the data is expected to look like, but for clustering to work well, the fewer assumptions that are made about the data the better it is. The algorithm should just be allowed to explore the interrelationships between different data points and conclude with the clustering results [17].

There are many clustering algorithms including partitioning methods, hierarchical methods, density based methods and grid-based methods. The one that is most relevant to this research is a partitioning method known as k-means.

5.4 k-Means Clustering

k-means also referred to as the Lloyd-Forgy method is one of the most popular clustering algorithms and dates back to the 1960's [18]. This is an unsupervised machine learning strategy which is extremely useful for cluster analysis. This is a centroid

based partitioning technique whose main aim is to cluster a set of data into a given number of k subsets which have high intra-cluster similarity and low inter-cluster similarity.

5.4.1 k-Means Algorithm

The algorithm for k-means requires an input of k which is the number of clusters and the set of data D which contains n data points. The output of this algorithm is the set of k clusters where each data point in D has a corresponding cluster number 0 through $k-1$. Each clusters center is the mean value of all the objects in that cluster. The algorithm is defined in the following steps: [15]

```
1 Randomly choose k data points from D as initial cluster centers
2 do{
3       (Re)assign each object to the closest cluster based on similarity
4       Update the cluster means by averaging points assigned to the cluster
5 }while(until no change);
```

This process is best explained through an example, consider Figure 5.1. In image (a) the initial data set D is shown. This is just what the initial data looks like, and for the first step of k-means 2 initial cluster centroids are picked, as shown in image (b). There are different variations of k-means where the initial cluster centroids are sample points within the data set D, or alternatively they are randomly generated points within the same space as the data. In this case it is the latter as shown in image (b). The next step is to look at the Euclidean distance between the centroids and each point and then assign each point to the closest centroid. This is shown in image (c) when all the points closer to the red centroid are marked as red and same for the blue points.

The next step is the recalculation of the centroid based on the mean of all the

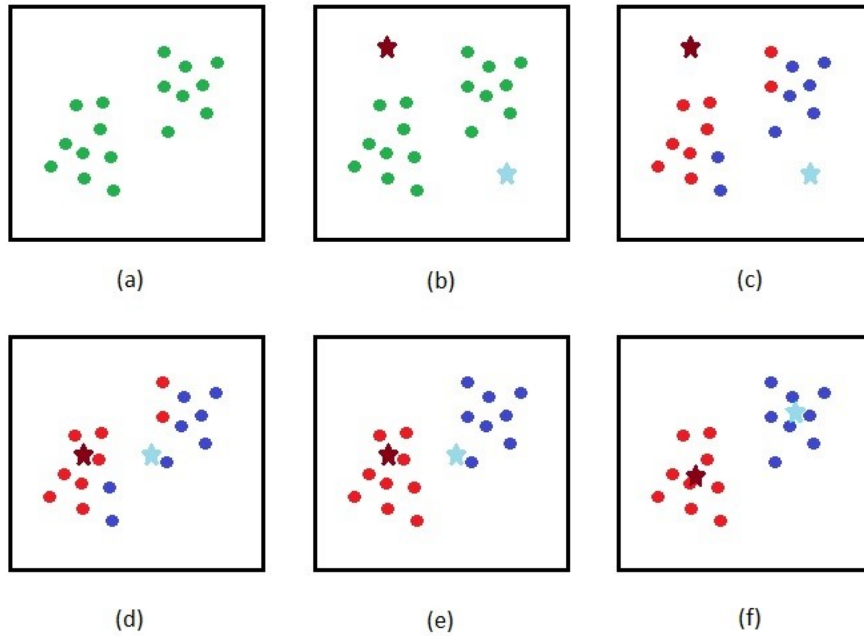

Figure 5.1: k-Means as a step by step process [19]

points that have been assigned to that cluster. As a result, an average centroid is taken of all the points marked as blue and red independently and the centroid is updated to that new location as shown in image (d). At this point the previous steps were repeated and then the individual points were reassigned based on the euclidean distance from the closest newly computed centroid location. In this step there is a recoloring of each of the points as either red or a blue as shown in image (e). Based on the newly clustered points, a new mean is calculated, and the centroids are repositioned to this newly computed mean value as shown in image (f). The algorithm terminates since no further change occurs.

5.4.2 *k*-Means Complexity Analysis

The k-means algorithm is widely used throughout this research project, so it is important to know the space and time complexity of this algorithm. The time complexity

of this algorithm would be $O(nki)$ where n is the size of the input dataset, k is the number of clusters that are chosen, and i is the number of iterations. i is directly proportional to n, and the number k is fixed and becomes a constant, making the effective time complexity $O(n^2)$ [20]. The space complexity is not very high since the dataset needs to be stored along with which cluster they belong to and the centroids of each cluster. Therefore the space complexity would be $O(n + n + k)$, here n is the dataset and k is the number of clusters. This simplifies to maintaining a space complexity of $O(n + k)$.

5.4.3 Limitations of *k*-Means Clustering

The input to the k-means algorithm is the number of clusters k. This value is fixed at the beginning and does not change through the entire process. This could be a potential issue if there are only 2 distinct clusters that are in the dataset and the value of k that is forced is 3. This could break apart one of the clusters and give a result which is skewed for more clusters than what is actually present in the data. It could go the other way as well, if there are more clusters than the number k that is provided, then the output results might not be as good. In this case clusters which should have separated out would have the same clustering value. The big limitation here is that the value k is not flexible at all. Choosing an optimized value of k is extremely important for this research, otherwise a forced incorrect answer would be forced on the specific dataset. This is a NP-hard problem and therefore a computationally accurate solution can be hard to find.

Another limitation is that the clusters that are formed are dependent on the original positions of the centroids. It is possible but unlikely to get stuck in a local minimum value and the algorithm might terminate before the results are optimized. This could occur if the initial centroids that were picked were outliers.

5.5 Clustering vs. Classification

Broadly speaking, clustering and classification solve different problems. An important part in deciding what method to consider would be the type of data available to the user. If the data contains an expected label result, then classification techniques can be used and an accuracy can be calculated based on that. However, in the absence of this data, clustering techniques would be ideal so that the data can be labeled based on a relative separation.

For this thesis, an important goal is to determine if process profiles are indicative of process behavior. In order to understand if this is possible, clustering based on process profiles will be used to determine how processes (executed applications) are grouped together. The hope is that the resulting clusters are indeed indicative of behavior at the desired level (usage class); therefore, the data lends itself more to a clustering approach than classification since the data only has an input and the correct output behavior is unknown.

The other important factor that needs to be kept in mind is that this research is trying to identify behavior, and a list of a finite number of behaviors is not necessary a priori. At this point this research is exploring a relatively new area and is user dependent as to what experiments need to be run, with a user chosen value of k. The drawback here is that the value of k cannot be predetermined and is based on the user choice. Optimizing this value of k using a computational technique would be ideal, and the silhouette coefficient method was chosen. This method provides a value for how compact and far from other clusters a certain cluster is. A silhouette coefficient is calculated for a range of values of k for the k-means method. The highest silhouette coefficient score would indicate that the cluster in set k which has that value is most likely the different number of behaviors found in that specific data set.

5.6 Silhouette Method

When considering clustering techniques, the cluster quality can be quantified based on how compact and separated the clusters are. A similarity metric referred to as the silhouette coefficient is a straight-forward method that can be used to measure the quality of the clusters [15].

There are two components of the silhouette method that need to be calculated. To calculate the silhouette score of a point, the first calculation needed is how compact the cluster is to the point. This is defined as $a(o)$ and is the average distance between o and every other data point which belongs to the same cluster as o. Ideally the value of $a(o)$ would be smaller indicating the compactness of the cluster. The second calculation is how separated o is from the closest clusters. This is defined as $b(o)$ and is the minimum average distance between o and points in other clusters. Ideally the value of $b(o)$ would be larger indicating that the separation between o and closest cluster is large. Both of these components are formally defined below: [15]

$$a(o) = \frac{\sum\limits_{o' \in C_j, o \neq o'} \text{dist}(o, o')}{|C_i| - 1} \tag{5.1}$$

$$b(o) = \min_{C_j : 1 \leq j \leq k, j \neq i} \frac{\sum\limits_{o' \in C_j} \text{dist}(o, o')}{|C_j} \tag{5.2}$$

The silhouette score of each point is then calculated by normalizing the difference between $a(o)$ and $b(o)$. This is done by dividing by the maximum of $a(o)$ and $b(o)$ and is referred to as $s(o)$.

$$s(o) = \frac{b(o) - a(o)}{max\{a(o), b(o)\}} \tag{5.3}$$

This value s(o) would contain the silhouette score of just the datapoint o and it would have a range from -1 to 1. To generalize this further an average could be taken across every single point in the dataset to give us quantitative readings of how this specific clustering scheme performed. [15]

A silhouette coefficient of close to 1 would indicate that the clusters are compact and far from other clusters so this would be ideal. If $b(o)$ less than $a(o)$, this means that the silhouette coefficient is negative indicating that the point o is closer to more points outside its cluster and further away from points within its own cluster. This would not be very desirable and would indicate that the clustering algorithm has not done as well as one would have hoped.

The silhouette coefficients can only be calculated once a clustering algorithm has been run on the data. It is known that one of the big limitations of k-means is that the value of k (the number of clusters) that is chosen is rigid. However with the combination of the silhouette coefficient, if a range of values of k are chosen then perhaps an ideal number k based on the silhouette scores can be identified. These scores are not available before an experiment is performed, so an ideal k can only be found experimentally. However this method combined with k-means could give stronger results than just k-means alone. This also lends itself to addressing the NP-hard problem that is finding an optimum value of k for k-means.

Chapter 6: Experiments

In this chapter, inferring process behavior is investigated experimentally using the k-means clustering on different process profiles. This chapter will cover three different sets of experiments that were run. It will go over the experiment flow and the toolchain for these experiments, the usage classes, and the impact command line arguments have on the behavior of an application. This is the initial work, Usage class group work, and then silhouette methods experiments.

The applications used for experiments (associating behavior with process profiles) were Linux commands, also known as Linux utilities. Using Linux commands as example applications has multiple advantages including its wide application and that the operations of these commands is widely known. There are also many Linux commands which have a general known range of different/similar functionality. Therefore based on these functions, it would be intuitive to predict what the behavior would be. There are also commands like `tar`, which based on the flags used, would change the behavior as detailed in Section 2.1.

6.1 Process Usage Classes

Broadly speaking, there are a range of applications which have different behavior. Specifically, for this thesis, Linux instructions are looked at since they are easily available and have a variety of different behaviors. Every application could be classified into a specific genre and by extension so could every Linux command. This would be based on what is expected from the command and have been divided into various usage classes. These usage classes are taken from previous work done in this area at Wake Forest University and are shown in Table 6.1 [5]. The descriptions and

an example of each of the usage classes is given below along with an example command in Linux which would fit this class. Many of the experiments are constructed to compare inter-usage classes and intra-usage classes.

Usage Class	**Example Linux Commands**	**General Class Description**
File system Interaction	`ls`, `find`	Accesses directory information
File display	`cat`, `head`, `tail`	Accesses the contents of a file
Encryption	`sha256sum`, `md5sum`	Creates a unique hash of a file
Compression	`gzip`, `bzip2`	Creates compressed files
Disk Management	`du`, `di`, `df`	Accesses disk space and usage
Process Management	`ps`, `top`	Accesses information related to running processes
Networking	`wget`, `curl`	Accesses resources over the internet to transfer files

Table 6.1: Process usage classes, their descriptions, and examples use for experiments.

6.2 Toolchain

The software tools developed for this thesis consist of several individual Linux-based programs that can be linked together via `bash` to form a toolchain. Figure 6.1 depicts the component of the toolchain.

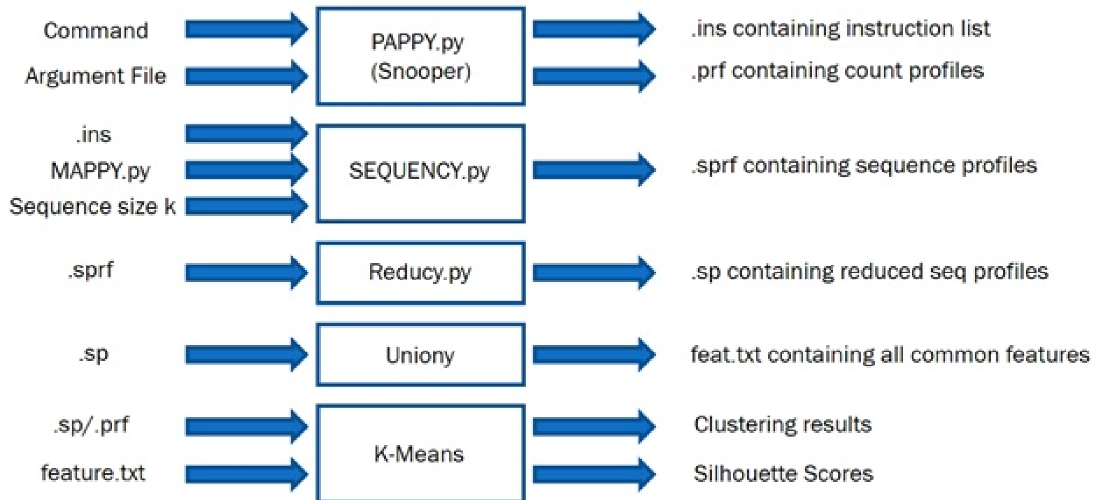

Figure 6.1: Overview of the tools and toolchain used for experiments.

PAPPY.py is a Python application that derives its name from Parallel Application and is the program that executes Snooper for a targeted process. This application outputs a list executed instructions (in the order they were executed) and an instruction profile. SEQUENCY.py provides sequence profiles using the instruction list from the Snooper output, a fixed sequence length of s, and MAPPY.py (which maps instructions, if requested) as imput. The output of SEQUENCY.py is the corresponding sequence profile of the input provided. Reducy.py and Uniony are Python and bash scripts, respectively, that operate on sequence profiles and remove all the features (instruction or sequence counts) that have frequencies of 0 and then align (renumber) the remaining features to maintain consistency. This is extremely useful when dealing with extremely large sequence sizes. Finally the k-means algorithm is performed on either the instruction profile or the sequence profile. Then the clustering results and the silhouette scores are produced as the output. Therefore an example sequence of operations could be PAPPY.py, then SEQUENCY.py (which also potentially invokes MAPPY.py), then Reducy.py, then Uniony, then finish with k-means. This sequence of applications could be easily written and executed as a bash or Python script.

6.3 The Effect of Command Line Arguments on Profiles

In this section, the impact of different command line arguments for the same application is explored. Consider an example of "File Display Usage Class." This includes commands like `cat`, `head`, and `more`. The command line argument could be a file, and the size of the file could vary significantly. This section will go over if varying file size impact the profiles created. Other examples are file system interaction usage class, which include commands like `ls` which displays the files in a directory. Therefore when this command is run with an argument of an empty directory vs one with a lot of files, will there be a difference in the process profiles produced? Multiple experiments were conducted to get answers to these questions.

Usage Class	Linux Command	Mean	Std. dev
File Display	`tail`	2086.8	225.6
	`cat`	427.0	0.0
	`head`	1955.8	225.6
File System Interaction	`find`	12970.7	197.8
	ls	1498.6	113.6
Disk Management	`du`	177852.0	216.1
	`df`	11510.0	216.1
Encryption	`md5sum`	450.0	0.0
	`sha256sum`	450.0	0.0

Table 6.2: Mean and standard deviation of the total sequence frequencies for various Linux command given 20 unique arguments.

Experimentation was done on 20 separate files and 20 different directories. There was a variation in the sizes of the files and the quantity of files in the directories as well. Sequence Profiles of size 4 with a mapping scheme of 3 were created, and the mean and standard deviation of the number of sequences can be found in table 6.2. This tells us that there are Linux commands like `cat`, `md5sum`, and `sha256sum` that always have a standard deviation of 0 and thus are command line argument independent, but the rest of the Linux Commands have a dependency on the provided argument. It can

also be seen that there is consistency in standard deviation within "Usage" classes. File display has a standard deviation of exactly 225.6 for both `tail` and `head`. "Disk Management" is at exactly 216.1, and Encryption is at 0. This consistency in standard deviation suggests a pattern even if the mean is very different.

6.4 Impact of Library Instructions on Clustering

As mentioned earlier, often the majority of the Snooper output consists of the machine instructions associated with library calls, and it would be very beneficial to see if behavior of different processes could still be distinguished without the library calls. k-means clustering was performed on pairs of profiles to see how well they can be distinguished based on these profiles. For example, consider the case of `head` and `tail`. Both of these commands are to display only the top or bottom of a file and belong to the same usage class. After constructing Instruction profiles, a confusion matrix was generated to see how well these clustered apart with and without library calls as shown in Figure 6.2.

Accuracy = 52.5%		Expected	
		head	tail
Predicted	head	20	0
	tail	19	1

With Library calls

Accuracy = 92.5%		Expected	
		head	tail
Predicted	head	20	0
	tail	3	17

Without Library calls

Figure 6.2: Confusion matrix for `head` and `tail` with and without library calls

The highlighted boxes are the boxes which accurately predicted what value was expected. The other values are a count of how many times the expected outcome did not align with the predicted outcome. The accuracy of the model is calculated by the sum of the highlighted regions divided by the sum of the entire matrix. According to

Exp No.	Command 1	Command 2	Accuracy with Library Calls	Accuracy without Library Calls
1	`ls`	`cat`	100.0%	100.0%
2	`ls`	`df`	100%	100%
3	`ls`	`sha256sum`	65%	70%
4	`sha256sum`	`cat`	67.50%	70%
5	`cat`	`df`	97.5%	97.5%
5	`tail`	`cat`	52.5%	100.0%
6	`sha256sum`	`md5sum`	100.0%	92.5%
7	`tail`	`head`	52.5%	92.5%

Table 6.3: Accuracy of different pairs of Unix commands with and without processing library calls.

the figure, it can be seen that the accuracy of this experiment with library calls vs. without library calls is significantly different.

There is an improvement from 52.5% to 92.5% when the library calls are removed. The results indicate that the instruction profiles that include library calls may not be reliably used to cluster the two commands into two distinct categories. In contrast, the omission of the library instructions does result in very accurate clustering. Therefore, it is possible that the inclusion of library instructions may simply obfuscate the details (adds too many instructions) of the commands. This is considered to be a harder problem since the profiles of head and tail belonging to the same usage class "File Display" and are expected to have similar profiles.

Many experiments were conducted but the most representative set was shown in Table 6.3. Experiments numbered 1-5 are inter-usage class experiments while 6-8 are within the same usage class. In almost every experiment listed there is either an improvement or no change in the accuracy when clustering two commands apart. `ls` vs. `cat` and `ls` vs. `df` have a 100% accuracy before and after removing library calls. Removing the library calls had no effect on accurately distinguishing behavior since

the profiles were already so different in these two cases but it is noteworthy that the accuracy was not decreasing.

In experiments 3 and 4 as shown in Table 6.3, it is observed that the removal of the library calls positively impacts the accuracy in behavior prediction. This is a 2.5% and a 5% increase in favor of removing the library calls. Therefore for intra-usage class there is a slight positive trend when library calls are removed. The benefit is twofold since there are far fewer instructions that need to be processed.

For the intra-usage class examples, consider the usage class of encryption which is experiment 7 in Table 6.3. Without library calls, this has an accuracy of 100% but after taking the library calls out there is a drop in the accuracy to 92.5%. So in this case, the profiles contained some amount of vital information in the form of library instructions which were lost leading to a drop in the accuracy of the clustering scheme. However without the library calls, there was still enough information to get an accuracy of 92.5%. The reason that `sha256sum` and `md5sum` may have not been correctly identified without the library calls could be because both of them might have very different library calls, and since they are both just creating a unique hash for a given file the application calls might be extremely similar. The behavior for this entire class is expected to be very similar and fall under the same class.

Removing the library calls will reduce the number of counts for each instruction encountered and also reduces the total number of unique instructions encountered. As shown in Table 3.4 the percentage of application based instructions is low and therefore by removing more than 99% of the data the process profiles would be significantly reduced. It is also worth noting that for the Linux command `ls`, on average there were 92.2 unique instructions when library calls were included and only 37 unique instructions without library calls. In conclusion, the removal of the library calls appears to be beneficial for both the accuracy in clustering utilities and even with significantly

reducing the feature space of profiles. Therefore all future experiments shown in this thesis will not include library calls.

This thesis is concerned with a broader identification of behavior. Therefore if there are failures within usage classes that is not as damaging. Another set of experiments was considered where multiple members of each usage class were considered. Since a broad behavior pattern is trying to be identified, this set would be more representative of behaviors as a class and is detailed in Section 6.5.

6.5 Clustering using Instruction Profiles

For this set of experiments, 4 usage classes were selected to analyze if the behavior of 2 commands within the usage class would model similar enough behavior to cluster together when compared with 2 commands from another usage class. The Instruction Profiles were constructed without library calls since the previous section (6.4) showed the added benefits that come with removing them. Some of the experiments are in Appendix C

This set of experiments could give us a good indication of what the potential labels of classification are and if they match with the usage classes. The usage classes that were chosen are "File Display, File System Interaction, Encryption" and "Disk Management." These experiments were run to show how well instruction profiles cluster out using k-means. The number of clusters for the experiments were ranged from 2 to 5 to see how the dataset cluster when different number of clusters are forced on the data.

6.5.1 File Display vs. File System Interaction

k (clusters)	[`cat`]	[`tail`]	[`ls`]	[`find`]
2	[1 1 1 1 1 1 1 1 1 1]	[1 1 1 1 1 1 1 1 1 1]	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]
3	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]	[2 1 2 2 2 2 1 1 1 1]	[2 2 2 2 2 2 1 2 1 1]
4	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]	[1 1 1 1 1 1 1 1 1 2]	[3 3 3 3 3 3 1 3 2 1]
5	[0 0 0 0 0 0 0 0 0 0]	[3 3 3 3 3 3 3 3 3 3]	[1 4 1 1 1 1 4 4 4 2]	[1 1 1 1 1 1 2 1 2 2]

Table 6.4: Clustering of File Display and File System Interaction groups with different number of targeted clusters.

The 4 vectors in table 6.4 have a length of 40. There are 4 different commands that are being tested, and there are 10 variations of each of them. In this example, the first 10 numbers represent the command `cat`, 11-20 represents `tail`, 21-30 represents `ls` and 31-40 represents `find`. These partitions are depicted with the [x x x x], where x refers to the cluster value k-means assigns to each profile. The numbers in the vector are dependent on the number of k clusters and represent the cluster to which each command is more closely associated with in terms of their behavior.

When $k = 2$ using k-means, it was found out that File display and File System interaction clustered separately. The first 20 values have a number 0 and 21-40 have a cluster number 1. This means when there are two possible categories, `cat` and `tail` would have similar enough behavior where they would cluster together, and the behavior would be different enough from `ls` and `find` that it would cluster separately from them. This solution shows that there is merit to the division of behavior by this set of usage classes.

When $k = 3$ and $k = 4$ then `head` and `tail` are labelled as 0 which means that they are more similar to each other than `ls` and `find`, and k-means still can not tell them apart. The commands `ls` and `find` have been divided into two categories first and then into three categories with no clear divide between the two. This could

indicate that the contents of the instruction profiles generated are extremely variable when using the find command since it can be seen when $k = 4$ that 9 out of 10 of the `ls` cluster together while the remaining find are broken into 3 separate clusters. This makes sense since the find command would have to display directories and sub-directories depending on the input. This could be a large output while an `ls` would simply always do a display of just the single directory. The extra step that the `find` profile has to take might lead to variability in the profiles that are generated, leading to some find profiles generated identical to `ls` and some generated differently.

When $k = 5$ `head` and `tail` can be distinguished. However looking more closely at just the 2nd half of each of the vectors which represent the `ls` and `find` command there is really no pattern as to how things are being clustered for $k = 3$ through 5. This could have to do with randomness in the selection of the starting centroids for the k-means algorithm.

6.5.2 Encryption vs. File System Interaction

k (clusters)	[`sha256sum`]	[`md5sum`]	[`ls`]	[`find`]
2	[1 1 1 1 1 1 1 1 1 1]	[1 1 1 1 1 1 1 1 1 1]	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]
3	[2 2 2 2 2 2 2 2 2 2]	[1 1 1 1 1 1 1 1 1 1]	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]
4	[1 1 1 1 1 1 1 1 1 1]	[2 2 2 2 2 2 2 2 2 2]	[3 3 3 3 3 3 3 3 3 3 3]	[0 0 0 0 0 0 3 0 3 3]
5	[2 2 2 2 2 2 2 2 2 2]	[1 1 1 1 1 1 1 1 1 1]	[0 3 0 0 0 0 3 0 0 3]	[4 4 4 4 4 4 3 4 3 3]

Table 6.5: Clustering of Encryption and File System Interaction groups with different number of targeted clusters.

According to Table 6.5, when the cluster size is limited to 2 the data for `sha256sum`, `md5sum`, `ls`, and `find` indicates that there is a clear distinction in the different usage classes of encryption and file system interaction. The behavior of these two broad groups is different enough, and a distinction can be made. When the cluster size is set to 3, `sha256sum` and `md5sum` are clustered apart since they are different enough while

`ls` and `find` can still not be distinguished. This means that `sha256sum` and `md5sum` are more different from each other than the entire file system interaction usage class.

When the cluster size is set to 4 and 5 the file system interaction usage class start to internally inconsistently cluster and there is no clean division between commands `ls` and `find`. This shows that both these commands are quite similar as described in section 6.5.1. To some extent for cluster size of 5, these two usage classes with 2 commands each can be considered to be clustered apart.

6.5.3 File Display vs. Disk Management

k (clusters)	[`cat`]	[`tail`]	[`du`]	[`df`]
2	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]	[1 1 1 1 1 1 1 1 1 1]	[0 0 0 0 0 0 0 0 0 0]
3	[1 1 1 1 1 1 1 1 1 1]	[1 1 1 1 1 1 1 1 1 1]	[0 0 0 0 0 0 0 0 0 0]	[2 2 2 2 2 2 2 2 2 2]
4	[3 3 3 3 3 3 3 3 3 3]	[2 2 2 2 2 2 2 2 2 2]	[1 1 1 1 1 1 1 1 1 1]	[0 0 0 0 0 0 0 0 0 0]
5	[3 3 3 3 3 3 3 3 3 3]	[2 2 2 2 2 2 2 2 2 2]	[1 1 1 1 1 1 4 1 1 1]	[0 0 0 0 0 0 0 0 0 0]

Table 6.6: Clustering of File Display and Disk Management groups with different number of targeted clusters.

To understand what is happening in this example with the cluster number limited to 2, there needs to be a slightly deeper understanding of how the `du` and `df` commands work. Since `df` and `du` are both employed to display the amount of disk space used, the expected disassembled code could be expected to be similar. However while both of these processes do things very differently, they are both measuring different things. The command `df` relies on system information which may or may not be outdated and gets information from files and quickly provides an approximate estimation of the disk usage. The command `du` relies on what can be seen at the particular instance it is run and will base its calculation on that. This command could be considered to be more complicated since it is multifaceted. The command `du` is the more accurate option but will take longer to compute. Rarely will `du` and `df` return the same value

since they are measuring two separate things. Interestingly the differences in behavior might cause them not to cluster together [21].

Since `du` does more than just display contents and does a more methodical search on what the disk data is, when a cluster size is limited to 2 the "Disk Management" usage class breaks apart. `du` clusters differently from `df` even though they should have the same expected output. This experiment really shows that clustering based on instruction profiles is sufficient to determine that, although the behavior is part of the same usage class and is expected to be the same, the underlying instructions performing the tasks are behaviorally different. The commands `cat` and `tail` should definitely cluster together, but that the behavior of `df` is more closely modeled to a `cat` and `tail` than a `du` is very telling of how the commands works.

When clusters are limited to 3, the disk management usage class breaks apart, and `du` and `df` both get their own cluster. This means that `cat` and `head` are closer to each other in terms of behavior than `df` which is why `df` will now cluster apart. This also consistent with expectations since they are in different usage classes. When the cluster size is set to 4, `head` and `tail` get distinguished, and all 4 of the utilities that are being tested are given their own cluster. Interestingly enough, this was not the case in the previous section when File Display was compared to File System Interaction. This reaffirms what was known about clustering which is that since the clusters identified are based only on the information provided, they are only relevant within the context of the information.

6.5.4 Experimental Analysis

It was demonstrated in the previous sections that certain classes of application behavior tend to cluster together using only instruction profiles (without library instructions) as features. There are three other file experiments which are part of this set

which are shown in the appendix item 3.

To some extent it can be said that usage classes do represent similar behaviour, and they are expected to cluster together. The outlier was `du` and `df` but it turns out that `du` was a far more complicated utility, and the behavior analysis using k-means was successful in identifying that. The experiments show a multitude of different clusters and how they perform given different usage classes. However identifying which cluster size is ideal and how things are separating apart becomes quite a challenge.

One of the biggest drawbacks of k-means is that it is very rigid when it comes to picking a value of k, and even if the ideal number of clusters is 2, if a cluster number of 5 is chosen then k-means will do its best to fit 5 clusters. This is where the silhouette coefficient becomes very useful. This metric as described in Section 5.6 gives a clustering result a numeric value between -1 and 1 which can be used to identify which cluster is the best.

6.6 Clustering using Sequence Profiles

The previous sections leveraged instruction profiles for clustering. In this section sequence profiles and mapping are used to cluster various applications (processes). In addition, the silhouette method is used to identify the best number of process clusters given a set of profiles. Furthermore, silhouette scores will be used as an indicator for the ideal sequence size and number of mapping categories. As detailed earlier, as a silhouette score for a particular cluster gets closer to a value of 1, the better the fit of the data is for that particular cluster number k. Each of these experiments has been performed 10 times and was found to be identical given random seeds which indicates that k-means could have successfully optimized these clusters.

6.6.1 Experiments varying Mapping Schemes

In this section, experiments will consider only sequences of length 4; however the instruction mapping will vary, and the effect on the silhouette score will be observed. The concept of mapping was been comprehensively described in a previous section. The general idea is to map instructions that perform essentially the same task to one common label. The benefit of mapping is a potential reduction in the alphabet size (number of possible instructions to consider), which allows for longer sequences. For this experiment 4 different mappings are considered. The mapping "intel28.map" maps every instruction to one of 28 labels. The mapping "intel20.map" maps every instruction to one of 20 labels. The mapping "intel4.map" maps every instruction to one of 4 labels, while "intel3.map" provides the most *aggressive* relabeling and maps every instruction to one of 3 labels.

Linux commands `ls` and `cat` are considered, as these two commands have different behaviors and have clustered apart in previous examples. The number of behaviors that would be expected is 2 (one per group) and the results of the silhouette coefficients for different mappings are given in Table 6.7.

Mapping	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
intel28.map	[0.931]	0.836	0.738	0.645	0.646
intel20.map	[0.934]	0.83	0.729	0.649	0.653
intel4.map	[0.938]	0.811	0.763	0.669	0.673
intel3.map	[0.941]	0.782	0.757	0.687	0.684

Table 6.7: Silhouette scores for `ls` vs. `cat` with sequences length of 4 and different mappings. Best scores are denoted with square brackets [].

In this example, it is known that there are two behavior types, and experimentally it is seen that the silhouette scores of a cluster $k = 2$ are more prominent regardless of the mapping scheme that was chosen. The highest silhouette score is marked with

square brackets, []. However it is interesting to notice that as the mapping reduces from categories of 28, 20, 4, to 3, there is an increase in the silhouette scores from 0.931, 0.934, 0.938 to 0.941 for the clustering associated with the best silhouette score. A higher score means a more compact cluster which is further away from other clusters. Even though this is a small increase in these scores, a better score would significantly help with categorizing behavior for similar instructions. The clusters are getting more distinct as the number of categories that are mapped to reduces, showing that for this experiment the mapping categories are inversely proportional to the silhouette scores. The labels of the clusters show that when there are 2 clusters, there is a correct identification of all the `cat` sequence profiles as `cat` and all of the `ls` sequence profiles are correctly identified as `ls`.

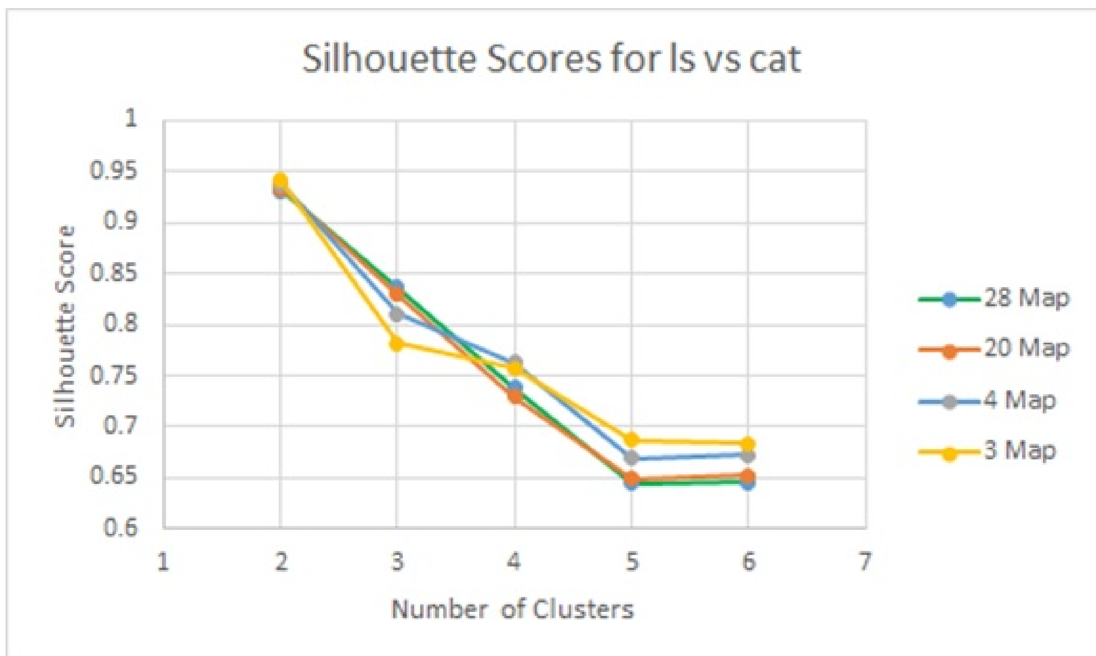

Figure 6.3: Silhouette scores for `ls` vs. `cat` with a sequence length of 4 with different mapping.

Figure 6.3 shows that the cluster 2 scores the highest across all the mapping schemes. This is the cluster that is expected to have the highest silhouette score since

there are two different commands from two different usage classes having different behaviors. Experimentally it can be seen that the 2nd highest silhouette score is consistently assigned to cluster 3 for each of the mapping schemes. What is interesting is the the cluster with the highest silhouette score, cluster 2, and that for the 2nd highest silhouette score, cluster 3, has the maximum difference when looking at a mapping scheme of 3. Therefore not only does the mapping scheme 3 has the highest silhouette coefficients as compared to the other mapping schemes it also creates the maximum distance between the highest scoring and the 2nd highest scoring clusters. This is a two fold advantage that mapping scheme of 3 has for this specific example. More experiments need to be conducted to see if this is a general trend for all of these cases.

Consider the usage class for "File System Interaction" vs "Encryption." The silhouette scores for this example are given in Table 6.8 using k-means with a value of k from 2 through 6. In this example it is interesting to see that once again a cluster size of 2 is consistently better than all the other cluster sizes. However the reduction in the number of categories that are being mapped to does not significantly impact the silhouette scores which are consistently quite high. This could also indicate the very clear separation between these usage classes

Mapping	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
intel28.map	[0.949]	0.849	0.861	0.834	0.770
intel20.map	[0.949]	0.848	0.861	0.833	0.769
intel4.map	[0.95]	0.857	0.865	0.839	0.779
intel3.map	[0.95]	0.854	0.862	0.835	0.788

Table 6.8: Silhouette scores for `ls`, `find`, `sha256sum` and `md5sum` with a sequence length of 4. Best scores are denoted with square brackets [].

These results indicate that varying the mapping does not have a significant neg-

ative effect on clustering; therefore, it may be possible to use more constrained maps (e.g. consisting of only 3 possible labels) which would also allow for larger sequences. Also there is still correct identification that cluster size of 2 has the best silhouette scores. In this example the difference between the highest scoring metric and the 2nd highest for all of the values of k is about the same but interestingly enough the top two clusters are $k = 2$ and 4. This is because 4 of these commands can cluster individually as well. The two examples that have been looked at so far are expected to have an ideal clustering of 2, and therefore silhouette scores were shown to work well to identify good clusters for a range of mappings. Here even the cluster size of 4 is more prominent as expected.

The next example is one that would have a silhouette score of 1. This would be multiple profiles of `ls`. A silhouette score of a cluster $k = 1$ would always be equal to $+1$ since there is no other cluster to compare with.The silhouette scores are given to us in Table 6.9. What can be observed is that $k = 2$ clusters always produces the highest result which is the closest to cluster 1 which would be ideal. There is also a downward trend as the number of k increases and moves away from the ideal answer $k = 1$.

Mapping	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
intel28.map	[0.883]	0.745	0.676	0.614	0.61
intel20.map	[0.887]	0.739	0.672	0.653	0.6
intel4.map	[0.896]	0.724	0.667	0.66	0.569
intel3.map	[0.902]	0.708	0.673	0.66	0.496

Table 6.9: Silhouette Scores for different profiles of `ls` with a sequence length of 4. Best scores are denoted with square brackets [].

These experiments serves as just one example of the types of experiments run. There were three types which were command vs command, usage class vs usage class

(2 commands each) and just one command. The latter was to show a downward trajectory as k moved further away from the ideal answer of $k = 1$. The first two types of experiments from this section are more relevant for the identification of behavior. In both these cases $k = 2$ had the highest silhouette scores and when smaller mapping schemes were considered, the silhouette scores were higher and further apart from the 2nd highest scores. Based on the experiments performed, more constrained mapping does not appear to negatively effect clustering performance (for Linux commands). The benefit of more constrained mapping is that it allows for longer instruction sequences.

6.6.2 Experiments varying Sequence Lengths

In this section a set of experiments was performed, where the mapping remained constant while the sequence length varied and their impact on the silhouette scores was observed. In this section a mapping scheme to three labels (intel3.map) was selected since this method seemed to outperform or do about the same as the other mapping schemes and similar experiments were run to calculate the silhouette scores for sequences of 2, 4, 8 and 16. Another benefit of choosing such a small mapping size is that it allows us to test sequences of size 16 without the complexity becoming too high. Let us consider the example `ls` vs. `cat`. This is expected to have 2 as the ideal number of clusters since the behaviors are different enough and belong to 2 separate usage classes.

Sequence Length	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
2	[0.944]	0.7950	0.661	0.711	0.738
4	[0.941]	0.782	0.757	0.687	0.684
8	[0.921]	0.851	0.756	0.752	0.661
16	[0.909]	0.868	0.793	0.79	0.766

Table 6.10: Silhouette scores of `ls` vs. `cat` with a mapping scheme of 3 (intel3.map) and different sequence lengths. Best scores are denoted with square brackets [].

From Table 6.10 it can be observed that as the sequence size increases the overall silhouette scores for the cluster $k = 2$ decreases. The decrease is from 0.944, 0.941, 0.921 to 0.909 indicates that a more more defined clustering occurs at smaller sequence sizes. This is not as significant as the difference between the cluster with the highest and 2nd highest cluster score. This difference is maximized for a sequence size of 4. As the sequence size increases, the difference between the highest and second highest silhouette scores gets smaller. For sequence of length 4 the difference is $0.941 - 0.782 = 0.159$ while for a sequence of length 16 the difference is $0.909 - 0.868 = 0.041$ which indicates that it is significantly easier to distinguish clustering happening with a sequence size of 4.

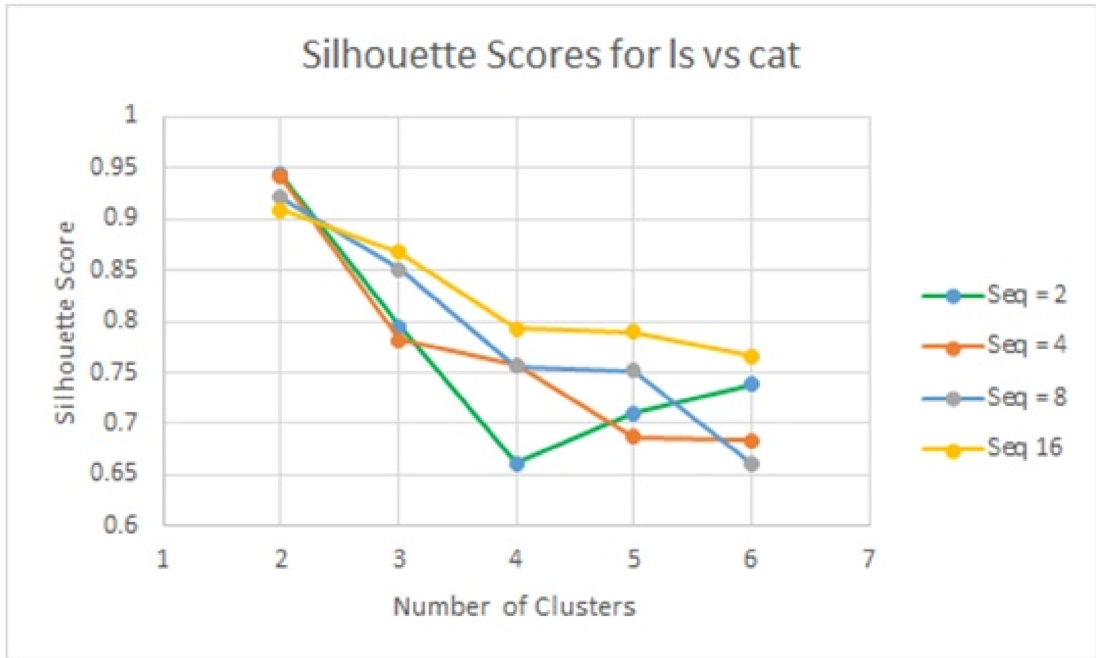

Figure 6.4: Silhouette scores for `ls` vs. `cat` with a mapping to 3 labels and a different sequence sizes.

As shown in Table 6.10 the behavior of `ls` and `cat` is correctly identified to have two distinct behaviors every single time. There is a relatively decreasing trend observed for all the clusters when the sequence size is varied except for sequence size of 2. This is easier to see in Figure 6.4 where the silhouette scores for a sequence size of 2 reaches a minimum at 4 cluster and then starts increasing. As the cluster sizes move away from 2, this decreasing trend should be observe.

The next example considers `ls, find, sha256sum` and `md5sum`. These are two separate usage classes and have clustered apart as 2 clusters in the past. The silhouette scores for cluster sizes 2 through 6 are given in Table 6.11

Sequence Length	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$
2	[0.952]	0.871	0.859	0.801	0.808
4	[0.95]	0.854	0.862	0.835	0.788
8	[0.939]	0.799	0.817	0.776	0.613
16	[0.93]	0.745	0.767	0.554	0.593

Table 6.11: Silhouette scores for `ls`, `find`, `sha256sum` and `md5sum` with a mapping scheme of 3 and different sequence lengths. Best scores are denoted with square brackets [].

A similar trend is observed with this experiment as well. As the sequence size increases the cluster with the highest silhouette score of size 2 shows a decrease in the silhouette score from 0.952 to 0.950 to 0.939 to 0.930. The sequence size of 2 seems across the clusters seems to have a decreasing and then increasing trend while the other sequences have a decreasing trend. It is interesting to see that the values of $k = 4$ (the number of targeted clusters) are slightly better than $k = 3$ and $k = 5$ except in the sequence size of 2. This is because there are 4 separate commands that are being tested and could be individually clustered apart leading to a silhouette score for 4 clusters to be higher. This perhaps indicates that sequences of 4, 8 and 16 are capturing that information but a sequence size of 2 does not. Therefore it can still be concluded that even though keeping the sequence size small would be ideal a sequence size of 2 is too small where valuable data is lost.

The last example is just a single `ls` command and the different silhouette scores are given in Table 6.12. It can be seen that as the sequence size increases the cluster size with the highest silhouette scores for each experiment decreases, and the difference between the winning cluster and the second highest cluster tend to have a maximum difference at sequence size of 4. But the difference between the best two silhouette scores gets smaller as the sequence size increases. Sequences of size 2 do not seem to perform as well and do not have this general decreasing trend as clusters increase for

each sequence which is present in all other sequence lengths.

Sequence Length	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
2	[0.909]	0.717	0.695	0.569	0.55
4	[0.902]	0.708	0.673	0.66	0.496
8	[0.864]	0.761	0.705	0.638	0.637
16	[0.839]	0.773	0.728	0.666	0.646

Table 6.12: Silhouette Scores for different profiles of `ls` with a mapping scheme of 3 with different sequence lengths. Best scores are denoted with square brackets [].

The broad conclusions that can be drawn from experimentation on varying sequence sizes is that as the sequence size decreases the silhouette score for clusters of size 2 tends to increase. This means that overall a decrease in this size would be more beneficial. Also the best silhouette score for an experiment tended to have a higher separation from the second best silhouette score; thus smaller sequences tended to generally perform best. Therefore it can be said that the smaller the sequence sizes the better the clustering for behavior analysis. These results indicate that a sequence of size 2 and 4 tend to do better than a sequence size of 8 and 16. However there is enough evidence to say that a sequence size of 2 is not actually capturing valuable data since it does not follow the general trends that are expected. Therefore maybe a sequence size of 2 is too small to capture integral sequential data. Therefore a sequence size of 4 which is the next smallest sequence size tested appears to be the best result.

Chapter 7: Conclusions and Future Work

The behavior of an application when it is being executed can be considered to be the general notion of what task the process is doing. Examples include displaying a web-page, printing a file, or computing statistics. Knowing the behavior of processes can be helpful for system administration. For example the allocation of computing resources, such as processor time, memory, and network access, can be tailored to improve efficiency and performance.

The main goal of this project was to infer the behavior of processes based on the machine instructions executed. The process profiles were of two types, the *instruction profile* and the *sequence profile*. Instruction profiles provide statistics about the individual instructions executed, while sequence profiles provide statistics about the sequences of instructions executed. For this thesis, the length of sequences ranged from 2 to 16 machine instructions. Generally speaking, sequence profiles are favored since the ordering of the data is preserved. The methods that were developed for analysis included the use of a k-means clustering algorithm and the silhouette method to determine the best clustering of applications based on different process profiles. The general method developed in this thesis had a very high success rate since it involved the actual running of the process to generate the profiles (dynamic analysis).

Experimentation was conducted to conclude that the removal of the library calls (such as reading and writing to devices) did not hinder the identifying of patterns of behavior. The removal of instructions associated with library calls helps focus on the actual code of the application which is crucial in identifying behavior. In some cases, this reduction was more than 99% since there are only a few application instructions.

This significantly reduces the dimensionality of the problem as well, by utilizing less than 1% of the instructions to still correctly identify usage classes/behavior. This shows the high levels of success with optimizing profiles which was another one of our goals.

The instruction profiles were successful in being able to distinguish between usage classes. There were multiple experiments to conclusively say that the usage classes that have been identified are good indicators of the behavior of the applications since similar applications clustered together and apart from dissimilar applications based on the *instruction profiles.* The exception to this was within the usage class Disk Management where the commands `du` and `df` were more dissimilar than expected.

The mapping schemes that were considered included 28, 20, 4, and 3 labels. Again, mapping is the process of assigning instructions that are functionally equivalent the same label. Experimentally it is shown that more constrained mapping (maps with fewer candidate labels) generally provided better clustering results. This indicates losing the specificity of Intel instructions helps infer process behavior. These differences in the actual instructions (those belonging to the same label) are largely unimportant for associating process behavior since they are perhaps largely the consequence of the programming language, programmer, or even the compiler. This also meant the sequence profiles generated with a mapping scheme of 3 labels would be smaller and also have better performance.

Despite the simplicity of instruction profiles, there are distinct benefits of sequence profiles. Experimentation was conducted on the ideal sequence size and it was concluded that as the number of sequences increases, the harder it becomes to distinguish behavior. What this meant was that a smaller sequence size needed to be chosen but the sequence size of 2 appeared unable to capture enough sequential information to perform well. Sequences of 4 mapped instructions tended to provide

the best performance, indicating that sequences are helpful for inferring behavior.

7.1 Future Work

This thesis' approach has been able to successfully distinguish certain types of behavior based on all executed applications' instructions; however, it may be possible to infer behavior with only a smaller portion of the instructions (a sample). The benefit would be the ability to infer behavior before the process has completed execution and could be used to develop tools that would be preventative and not reactive. Looking at a certain percentage of instructions or a certain fixed number could be enough to distinguish behavior. There are other kinds of sampling techniques which can be applied to this research as well. Perhaps looking at alternate instructions or randomly selection 1 in every n number of instructions. This methods would reduce the feature space by a factor of n.

The toolchain (set of programs used to obtain experimental results) could be modified, and other techniques could be applied, allowing the performance could be compared. Instead of k-means clustering algorithms, a k-hierarchical algorithm could be used. Instead of Euclidean distance measures that are in place, a Manhattan distance could be applied. Currently only either the instruction profile or the sequence profile is used to distinguish types of behavior, but features of one could be added to the other to create a more distinct profile for behavior analysis.

Now that the initial work has been conducted using single-threaded applications, another direction would be to change Snooper to handle multi-threaded applications and create new parallel profiles. There is also a lot of scope for using the data that at this point to build a classifier. New labels could be invented, and there are numerous classification techniques which can be explored.

Bibliography

[1] Younge, Andrew J., et al. "Efficient resource management for cloud computing environments." International Conference on Green Computing. IEEE, 2010.

[2] How to use Tar Command in Linux with examples. (n.d.). From: https://www.interserver.net/tips/kb/use-tar-command-linux-examples/ Accessed on March 31st 2019

[3] Ishrat, Mohd, Manish Saxena, and Mohd Alamgir. "Comparison of static and dynamic analysis for runtime monitoring." International Journal of Computer Science & Communication Networks 2.5 (2012).

[4] Moser, Andreas, Christopher Kruegel, and Engin Kirda. "Limits of static analysis for malware detection." Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007). IEEE, 2007.

[5] Gupta, Charchil. Dynamic Analysis of Program Execution to Discover Usage Classes. Wake Forest University , 2017.

[6] The von Neumann Computer Model (n.d). From: http://www.c-jump.com/CIS77/CPU/VonNeumann/index.html Accessed on March 31st 2019

[7] Gorry Fairhurst. Example of Assembly. From: http://www.erg.abdn.ac.uk/users/gorry/eg2069/assembly.html Accessed on March 31st 2019.

[8] Arian Stolwijk. Dynamic disassembling instructions with ptrace and udis86 for timing analysis. From: http://www.aryweb.nl/2013/05/25/ptrace-timing-analysis-by-disassembling/ Accessed on March 31st 2019.

[9] Leslie, Christina, Eleazar Eskin, and William Stafford Noble. "The spectrum kernel: A string kernel for SVM protein classification." Biocomputing 2002. 2001. 564-575.

[10] Fulp, Errin W., Glenn A. Fink, and Jereme N. Haack. "Predicting Computer System Failures Using Support Vector Machines." WASL 8 (2008): 5-5.

[11] Mapping scheme for x86 instructions was inspired from: http://ref.x86asm.net/geek.html#x0F21 Accessed on April 5th

[12] x86 Historical overview From: https://www.computerworld.com/article/2535037/computer-hardware-happy-birthday-x86-an-industry-standard-turns-30.html Accessed on April 7th 2019

[13] Lison, Pierre. "An introduction to machine learning." (2015).

[14] Zoubin Ghahramani. Unsupervised Learning Lecture Notes. 2005. From: http://mlg.eng.cam.ac.uk/zoubin/course05/ Accessed on April 9th, 2019.

[15] Han, Jiawei, Jian Pei, and Micheline Kamber. Data mining: concepts and techniques. Elsevier, 2011.

[16] StatSoft, Inc. (2013). Electronic Statistics Textbook. Tulsa, OK: StatSoft. From: http://www.statsoft.com/textbook/ Accessed on April 9th, 2019

[17] Jain, Anil K., M. Narasimha Murty, and Patrick J. Flynn. "Data clustering: a review." ACM computing surveys (CSUR) 31.3 (1999): 264-323.

[18] E.W. Forgy (1965). "Cluster analysis of multivariate data: efficiency versus interpretability of classifications". Biometrics.

[19] Chris Piech, Andrew Ng. K Means Stanford CSC221. From: http://stanford.edu/ cpiech/cs221/handouts/kmeans.html Accessed on April 9th, 2019

[20] Pakhira, Malay K. "A linear time-complexity k-means algorithm using cluster shifting." 2014 International Conference on Computational Intelligence and Communication Networks. IEEE, 2014.

[21] Mike Golvach Why DU and DF display different values on Linux and Unix http://linuxshellaccount.blogspot.com/2008/12/why-du-and-df-display-different-values.html Accessed on April 16th, 2019

Appendix A: Instruction Profile

Instruction Profile of Linux Command *ls* without library calls

41 add
15 and
65 call
2 cmovnz
2 cmovz
120 cmp
1 div
15 ja
13 jae
13 jb
3 jbe
2 jg
1 jle
135 jmp
59 jnz
4 js
70 jz
24 lea
371 mov
2 movsxd
27 movzx
2 nop
1 or
66 pop
124 push
23 rep
1 repe
23 ret
3 sar
2 sbb
38 setnz
1 setz
4 shl
7 shr
22 sub
86 test
50 xor

Appendix B: Mapping Scheme Details

Intel map 4		Intel map 3	
No. of ins.	Label	No. of ins.	Label
153	ARITH	162	ARITH
9	ARITH-STRING	7	BIT
7	BIT	84	BRANCH
84	BRANCH	47	COMPAR
5	BRANCH-STACK	46	CONTROL
7	BREAK-STACK	54	CONVER
47	COMPAR	161	DATAMOV
46	CONTROL	13	FLGCTRL
54	CONVER	10	INOUT
131	DATAMOV	1	INVALID
8	DATAMOV-ARITH	7	LDCONST
2	DATAMOV-FLGCTRL	21	LOGICAL
5	DATAMOV-SEGREG	7	SEGREG
15	DATAMOV-STRING	20	SHFTROT
13	FLGCTRL	15	SHIFT
2	INOUT	25	SHUNPCK
8	INOUT-STRING	26	STACK
1	INVALID	6	STRING
7	LDCONST	8	TRANS
21	LOGICAL	1	X87FPU
7	SEGREG		
20	SHFTROT		
15	SHIFT		
25	SHUNPCK		
8	STACK		
6	STACK-FLGCTRL		
6	STRING		
8	TRANS		
1	X87FPU		

Appendix C: Instruction Profiles Usage Class Experiments

k (clusters)	[cat]	[tail]	[sha256sum]	[md5sum]
2	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]	[1 1 1 1 1 1 1 1 1 1]	[1 1 1 1 1 1 1 1 1 1]
3	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]	[2 2 2 2 2 2 2 2 2 2]	[1 1 1 1 1 1 1 1 1 1]
4	[0 0 0 0 0 0 0 0 0 0]	[3 3 3 3 3 3 3 3 3 3]	[1 1 1 1 1 1 1 1 1 2]	[2 2 2 2 2 2 2 2 2 2]
5	[3 3 3 3 3 3 3 3 3 3]	[0 0 0 0 0 0 0 0 0 0]	[1 1 1 1 1 1 1 1 1 1]	[2 4 4 2 2 4 2 4 2 2]

Table C.1: Clustering of File Display and Encryption groups with different number of targeted clusters.

k (clusters)	[ls]	[find]	[du]	[df]
2	[0 0 0 0 0 0 0 0 0 1]	[0 0 0 0 0 0 0 0 0 0]	[1 1 1 1 1 1 1 1 1 1]	[0 0 0 0 0 0 0 0 0 0]
3	[0 2 0 0 0 0 2 2 2 2]	[0 0 0 0 0 0 2 0 2 2]	[1 1 1 1 1 1 1 1 1 1]	[0 0 0 0 0 0 0 0 0 0]
4	[0 0 0 0 0 0 0 0 0 0]	[3 2 2 2 2 2 3 2 3 3]	[1 1 1 1 1 1 1 1 1 2]	[2 2 2 2 2 2 2 2 2 2]
5	[4 4 4 4 4 4 4 4 4 4]	[0 3 3 3 3 3 0 3 0 0]	[2 2 2 2 2 2 2 2 2 2]	[1 1 1 1 1 1 1 1 1 1]

Table C.2: Clustering of File System Interaction and Disk Management groups with different number of targeted clusters.

k (clusters)	[ls]	[find]	[du]	[df]
2	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]	[1 1 1 1 1 1 1 1 1 1]	[0 0 0 0 0 0 0 0 0 0]
3	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]	[2 2 2 2 2 2 2 2 2 2]	[1 1 1 1 1 1 1 1 1 1]
4	[3 3 3 3 3 3 3 3 3 3]	[0 0 0 0 0 0 0 0 0 0]	[2 2 2 2 2 2 2 2 2 2]	[1 1 1 1 1 1 1 1 1 1]
5	[0 0 0 0 0 0 0 0 0 0]	[0 0 0 0 0 0 0 0 0 0]	[2 2 2 2 2 2 4 2 2 2]	[1 1 1 1 1 1 1 1 1 1]

Table C.3: Clustering of Encryption and Disk Management groups with different number of targeted clusters.

Arnav Bhandari

https://www.linkedin.com/in/arnavbhandari07 | bhana13@wfu.edu | 914.318.0689

EDUCATION

Wake Forest University (Graduate School of Arts and Sciences) Winston-Salem, NC
Master in Computer Science *August 2019*
GPA: 3.88
Defended Thesis Titled: Identification of Application Behavior Using Process Profiles
Relevant Coursework: Advanced Security, Theory of Computation, Theory of Algorithms, Databases and Software Engineering, Compilers and Translators, Digital Media

Wake Forest University Winston-Salem, NC
Major: Bachelor of Science in Computer Science, Minor: Mathematics *May 2017*
CGPA: 3.66, *Major GPA:* 3.82
Relevant Coursework: Genetic Algorithms, Machine Learning, Parallel Programming, Compilers, Bioinformatics, Codes and Cryptography, Computer Organization, Computer Systems, and Data Structures and Algorithms I and II

Honor Societies

Upsilon Pi Epsilon (Excellence in Computer Science)
Pi Mu Epsilon (Excellence in Mathematics)

SKILLS

Technical Skills: MATLAB, C/C++, Java, SQL, HTML, Python

PROFESSIONAL EXPERIENCE

Clinical and Translational Sciences Institute, Wake Forest Baptist Health
Winston-Salem, NC
Analyst 2, Programmer Intern *June 2019-Present*

- Created de-identification models using Natural Language Processing to protect PHI data with an accuracy of 97% using MIST software
- Used Natural Language Processing (NLP) tool: cTAKES to extract medical classifications (icd10 codes) from clinical data

Network Security Lab | Department of Computer Science, Wake Forest University
Winston-Salem, NC
Research Assistant for Thesis *May 2018-Present*

- Managed tools in Python for dynamic analysis to capture a programs behavior by observing machine-level instructions through process profiles
- Run simulations to distinguish, cluster and classify different basic instructions in Linux based on mapping and sequences

Biophysics Lab | Physics Department, Wake Forest University
Winston-Salem, NC
Research Assistant *August 2017-Present*

- Develop MATLAB code for classification of virus particle movements using variational Bayes and Hidden Markov Models
- Generated code for virtual data particle motion, a new batch processing and file management system, and data input output
- Utilize software for Image Processing including ImageJ and VideoSpotTracker to track x and y coordinates for individual virus particles
- Constructed MATLAB code to identify relative location of virus particle from the center of the cell and created a graphic for same

Computer Science Department, Wake Forest University
Winston-Salem, NC
Teaching Assistant *August 2016 -May 2019*

- Facilitate with Introductory Computer Science (Java and C++), Programming Languages and Data Structures and Algorithms II
- Teach over 100 students by problem solving and debugging code segments in multiple in-class labs (C++ and Java)

Undergraduate Research and Creative Activities Center, Wake Forest University
Winston-Salem, NC
Wake Forest Research Fellow *May 2016-October 2016*

- Collaborated with University of New Mexico and Wake Forest University on a Rotating Point Spread Function research project
- Created MATLAB code for computational techniques to convert 2D image data into a 3D space for mapping point sources
- Achieved a residual of less than 2.5 units of misfocus for 90.87% of the data and a residual of less than 1 for 75.93% of the data
- Presented my research poster at the Annual Undergraduate Research Day at Wake Forest

Mathematics Department, Wake Forest University
Winston-Salem, NC
Tutor *January 2015-May 2016*

- Applied knowledge of Discrete Mathematics and Calculus I and II to aid over 50 students with these courses and classwork
- Met with students on a one-on-one basis to answer any questions and taught them strategies to better improve their coursework

Investment Banking, Barclays Bank
Mumbai, India
Analyst Intern *June 2014-August 2014*

- Worked on peer bench marking for Mergers and Acquisitions and utilized Bloomberg software for equity analysis
- Created mathematical models using Excel for predicting earnings and performing growth analysis for various clients

LEADERSHIP EXPERIENCE

Association for Computing Machinery (ACM), Wake Forest University
Winston-Salem, NC
Treasurer, Vice President *August 2015-May 2017*

- Founded a raspberry pi initiative with a staff member to teach 10 interested students how to use this device in a 5 week course
- Maintained and balanced a budget for all ACM sponsored events in coordination with the Computer Science Department

South Asian Student Association (SASA), Wake Forest University
Winston-Salem, NC
Treasurer, Vice President *August 2015-May 2017*

- Collaborated with other organizations to organize large-scale events including Diwali & Eid, Holi for more than 400 students
- Fostered a sense of community and belonging for a minority on campus through programming and smaller social events