Techniques for the Abstraction of System Call Traces to Facilitate the Understanding of the Behavioural Aspects of the Linux Kernel

Waseem Fadel

A Thesis

In

The Department

Of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montreal, Quebec, Canada

November 2010

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Waseem Fadel

Entitled: Techniques for the Abstraction of System Call Traces to Facilitate the Understanding of the Behavioural Aspects of the Linux Kernel

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical Engineering and Computer Science) at Concordia University

Complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Dongyu Qiu
_____Chair

Dr. Nawwaf Kharma
_____Examiner

Dr. Peter Grogono
_____Examiner

Dr. Abdelwahab Hamou-Lhadj
_____Supervisor

Approved by:

_____
Chair of Department or Graduate Program

Director

_____20__ _____
Dean of Faculty

ABSTRACT

Techniques for the Abstraction of System Call Traces to Facilitate the Understanding of the Behavioural Aspects of the Linux Kernel

Waseem Fadel

Understanding the dynamic aspects of the Linux kernel can help in a number of software engineering activities including maintenance and program comprehension, performance analysis, and most recently security.

Dynamic analysis of the Linux kernel is accomplished by instrumenting the kernel and studying the generated traces. However, the major concerns that developers face when using dynamic analysis tools are the large size of the generated traces, and the low-level nature of their events.

In this thesis, we apply pattern detection and utility removal techniques on low-level system call traces generated from the Linux kernel. As a result, we obtain high-level abstracted traces that are more compact and readable, while still preserving the system main behaviour.

We apply our techniques to five different systems running on the Linux kernel and assess the effectiveness of our approach in terms of quantity where we measure the compression ratio and in terms of quality where we study how the high-level abstractions can convey more meaningful information about the program being traced than low-level system call traces.

Acknowledgements

I would like to take the chance to thank my supervisor, Professor Abdelwahab Hamou-Lhadj, not only for directing me through my work, but also for giving me his precious advices which definitely helped me through my stay and study in Canada.

Special thanks to Mario Couture from DRDC (Defence R&D Canada), Dominique Toupin from Ericsson Canada, and Dr. Michel Dagenais and his team (especially Pierre-Marc Fournier and Mathieu Desnoyers) from l'École Polytechnique de Montréal for their continuous support and feedback throughout this project.

I would also like to thank my friends at the lab with whom I had a lot of discussions which broadened my mind and enriched my knowledge of the field of software comprehension and maintenance.

Finally, I would like to thank my wife who without her I would not have been able to go all the way through this, and my family in Syria who supported me in all the means and encouraged me to do this important step of my life.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1. Problem and Motivations

Dynamic analysis is widely used by software engineers and developers to study system behaviour for many purposes like detecting bugs, understanding how a certain feature is implemented, reasoning about performance, or monitoring the system during its execution for detecting anomalies [1, 2, 3, 5, 6, 12, 16, 17, 18, 19, 33, 34].

Dynamic analysis is performed by executing a system and generating execution traces for the system under study through a process called instrumentation [19]. Instrumentation can be applied to the system by using probes (e.g. print statements) or through the control of a debugger. It may take place at different levels of abstraction: the source code, the binary code, the virtual machine, or the operating system. It could also be applied to generate information from the user space where routine calls are traced, or from the kernel space where lower-level system calls are traced [19].

Kernel space is the memory area where the kernel code is loaded and executed, and it is usually protected from any unauthorized access [7]. User space, on the other hand, is the memory area where user processes are loaded and executed [7]. User processes usually access kernel code by requesting services through calls to special functions known as system calls which provide "interfaces between User Mode processes and hardware devices" [8]. When tracing user space, we obtain the list of processes being executed and their internal routine calls. However, it is important to know how these processes interact with the operating system and what system calls they are

requesting. This can be accomplished by tracing the kernel space, where the resulting traces take usually the form of system calls and other low-level events.

Analyzing execution traces, however, can be a tedious task due to the large amount of data generated during run-time [1, 2, 3, 4, 5, 6, 16, 17, 18, 19]. The size problem becomes worse in the case of kernel space tracing due to the low-level events that appear in the trace such as memory management events, hardware interrupts, software traps, page faults, etc.

The objective of this thesis is to facilitate the analysis of large kernel space traces by developing techniques to reduce the size of traces. Our techniques are based on extracting higher-level information from low-level system call events. For instance, a sequential file reading operation may be the abstract representation of numerous, possibly out of order, disk block reads. We present a new trace abstraction algorithm based on pattern detection techniques. We also show the effectiveness of our approach by applying to traces generated from different systems.

1.2. Research Contributions

The main contributions of this research are summarized in what follows:

- A pattern library for Linux kernel system calls which models the most common operations of the Linux kernel. These operations include: File Management (Open, Read, Write, Close. Access, Stat), Socket Management for TCP and UDP (Create, Bind, Connect, Listen, Accept, Send, Receive, Close), Process Management (Clone, Execute, Exit), Memory Management and Page Faults.

- A trace abstraction algorithm that takes system-call traces as input and uses the pattern library to detect different patterns in the trace and replace them with higher level constructs that make the trace easier to understand.

- A utility pattern library that models noise in system-call traces such as memory management operations and page faults events.

- An experimental study that shows the effectiveness of the abstraction algorithm developed in this thesis.

- A tool that supports the abstraction process presented in this thesis.

1.3. Thesis Outline

The rest of this thesis is organized as follows:

- In Chapter 2, we present the background and related topics needed to understand the concept presented in this thesis. We present the most common abstraction techniques found in the literature to abstract and summarize execution traces.

- In Chapter 3, we introduce our approach to abstract out system call traces. First, we explain the system calls mechanism. Then, we present the patterns library that we have developed to characterize the Linux kernel operations. After that, the abstraction process is explained. Finally, we discuss the applicability of our approach to different kinds of processes.

- In Chapter 4, we evaluate our approach by applying it to four target systems. The results are discussed both from the quantitative and qualitative perspectives.

- We conclude our work in Chapter 5, and provide a summary of the main contributions and future directions.

Chapter 2. Background

There exist several studies that focus on reducing the size of large traces to facilitate their analysis. These techniques, however, have long focused on user space traces such as routine call traces [39]. These techniques vary in their design and can be grouped into three main categories: Sequence detection, summarization, and visualization. It should be mentioned that some techniques may belong to more than one category. For example, some trace visualization tools implement a number of sequence detection and summarization methods.

2.1. Sequence Detection

The sequence detection techniques are used to reduce trace size without losing information. This is possible due to the fact that many events are repeated in the trace as recurrent patterns or as a result of executing a loop or a recursive function. The techniques reviewed under this category follow similar strategies that focus on detecting such sequences and patterns and referring to them only once, while preserving the number of times and the order through which they appear in the trace.

Hamou-Lhadj and Lethbridge in [17] classified sequences with respect to their complexity level into three types: simple sequences, non-overlapping contiguous sequences, and overlapping non-contiguous sequences. Figure 1 shows how a contiguous repetition of the node B is replaced with the node SEQ (4) showing that B is repeated 4 times.

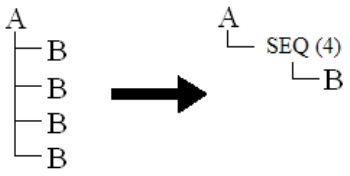

Figure 1. A sequence replaced with a node containing the number of occurrences

With simple sequences, events appear in the trace as repeated contiguous calls which might result from the execution of a loop or a recursive routine. Hamou-Lhadj and Lethbridge [17] proposed to remove repetitions due to loops and recursions in traces of routine calls and replace them with simple nodes. Figure 2 shows how a large trace can be turned into a more compact trace if this simple repetition removal technique is used.

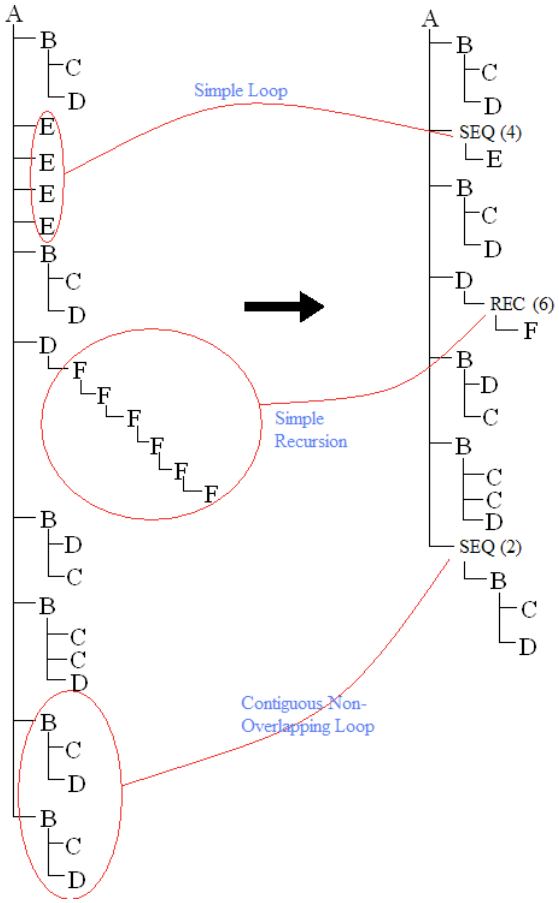

Figure 2. Detecting simple and non-overlapping Contiguous Sequences

In [17], the same authors propose a better trace reduction mechanism by detecting and representing only once sequences of events that are repeated multiple times but in different places. For example, we can see in Figure 2 that the sequence “ABC” is repeated multiple times in a non-contiguous manner. This sequence is detected using the concept of the common subexpression problem [17, 21] which consists of detecting similar subexpression in a tree structure. This is achieved by transforming a tree into an ordered directed acyclic graph (DAG). For example, in Figure 2 the trace is transformed into the graph of Figure 3. We can see that there is a gain in terms of size obtained through this transformation.

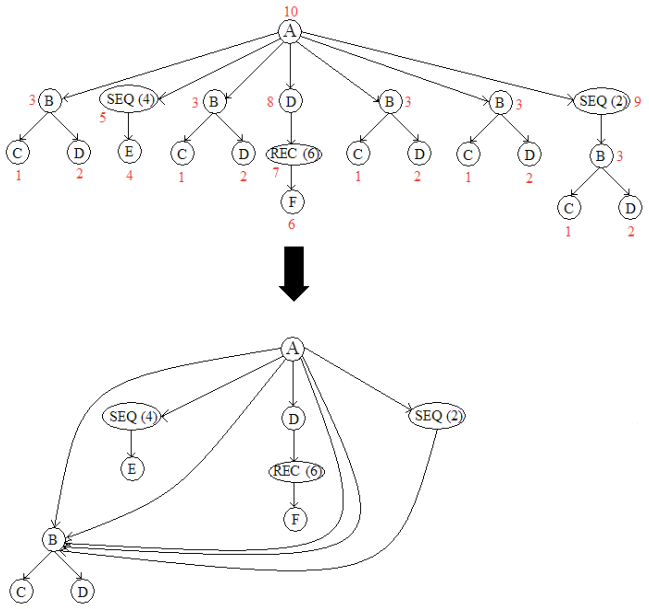

Figure 3. The trace after detecting non-contiguous sequences

The transformation of a tree into a graph can only result in high compression if some sort of matching criteria are used during the matching of the subtrees. For example two sequences ABBBCD and ABCD should be considered similar if we ignore the number of repetitions of B. This will further reduce the size of the resulting DAG. Hamou-Lhadj [19] introduced some matching criteria to generalize the algorithm in order to obtain better compression ratios.

2.2. Trace Summarization Techniques

Summarization techniques are used to extract high-level summaries from large traces. This allows software engineers to understand the big picture first before they decide to dive into the details. Many studies [1, 2, 18, 20, 23, 24] have been conducted to develop such techniques, relying on the fact that not all the information included in the trace are of the same importance to the developer, and that filtering the trace by removing low-level details would help generalizing it, and hence making the analysis process much easier.

Hamou-Lhadj and Lethbridge proposed an approach for extracting summaries from large trace that rely on the removal of implementation details [18]. They define an implementation detail as: "any element of a program whose presence could be suppressed without reducing the overall comprehensibility of the design of a particular feature, component or algorithm" [18]. Based on this definition, they proposed that several routines such as constructors, setting and accessing methods, private and protected methods and some user defined methods can be recognized as implementation details that could be detected and removed. They have also developed an approach for removing other utility components that cannot be grouped into these categories by studying their behaviour in the program [18, 20]. For example, a method with a high fan-in, i.e. being called in many different places in the program, is more likely to be a utility method [18] than a method with a low fan-in but a high fan-out, which must be doing something important since it generates several other calls. The authors have developed a utilityhood metric to measure the extent to which a method could be considered as a utility. Their summarization process uses this utilityhood metric to rank routines invoked in a trace and only keep the ones that are deemed non-

utilities. Several thresholds have been used. The results of their experiments are very promising.

Another interesting approach for trace summarization is the one proposed by Kuhn and Greevy in [2]. The authors introduced a technique by making an analogy between traces and signals. This analogy is based on the fact that a trace "is composed of monotone subsequences separated by pointwise discontinuities" [2]. Therefore, starting from the first event, a group is built by adding events with the same or with higher nesting level. However, when facing a decrease greater than a certain threshold known as gap size, a new group is created [2]. As a result, they were able to obtain a summarized trace which is only 10% of the size of the original trace.

Sampling [1] is another technique used to extract summaries from traces. It consists of selecting a sampled set of events from the trace. According to Chan et al., these events are selected in two ways: by considering each N^{th} event, or each N^{th} timestamp [1]. Depending on which variation to use, N could be defined as the set of the numbers of events to be selected, and could be computed as the following: N=S/M, where S is the total trace size and M is the maximum output size which is usually defined by the user [22]. On the other hand, N could be defined as the moments in time in which the events are selected [1].

Cornelissen et al. used a stack-based abstraction technique that summarizes the trace by removing certain events from the routine call stack [23]. They defined two variations of their approach. The first is the maximum stack depth, where events with nesting levels higher than a defined threshold are omitted, causing the removal of low-level detailed messages [23]. The second is the minimum stack depth, where

events with nesting levels lower than a defined threshold are omitted, removing high-level control messages that start up a certain scenario [23].

Nevill-Mannin and Witten developed an algorithm that generates a context free grammar from "a sequence of discrete symbols by replacing repeated phrases with a grammatical rule that generates the phrase, and continuing this process recursively" [24]. They applied their algorithm to compress million-symbol sequences representing biological objects, multi-megabyte DNA sequences, and identifying the structure of a database and compressing it [24]. The algorithm works by first replacing a sequence with a non terminal, and then it proceeds with reading the sequence to detect repeated phrases. When this happens, it generates a new rule with the repeated phrase on the right hand side and a new non-terminal on the left hand side, and then it replaces every occurrence of that phrase with the new non-terminal.

Later on, Larus improved this algorithm to compress execution traces during his work on defining and generating whole program paths that helps capturing program dynamic control flow [4]. The compression ratio is defined by the fact that (Log N) number of rules in the grammar can generate a number of N occurrences of a subsequence [4]. Unfortunately, this technique has two main drawbacks. The first is that it does not preserve the order of the events of a trace. The second drawback is that it does not preserve the number of occurrences of each of these events. However, it is useful in terms of extracting the hierarchical structure of the trace, and providing the developer with a summary of that trace.

2.3. Visualization Techniques

Visualization techniques have been helpful in reducing the trace size by allowing users to navigate through traces in an efficient manner. Through their work in [25], Bennett et al. grouped the features implemented in several visualization tools into two groups: Presentation features and interaction features. They defined the former as the set of features affecting the layout of the diagram, such as showing multiple views, hiding information and using animation.

Interaction features, on the other hand, are those implemented to enable users to interact with the tool by navigating, querying, and manipulating [25]. Example of such features include selecting trace content, providing the ability to navigate between components and instances, providing techniques such as: collapsing, partitioning, showing related messages to selected objects only, and single-step animation, zooming and Scrolling, etc. The following table summarizes trace abstraction techniques.

Table 1. Trace Abstraction Techniques

Approach	Techniques used
Sequence Detection	• Simple sequences • Non-overlapping contiguous sequences • Overlapping non-contiguous sequences
Trace Summarization	• Removal of implementation details • Grouping • Sampling • Stack depth limitation • Context free grammar
Visualization	• Multiple linked views • Hiding information • Zooming • Scrolling • Filtering • Single-step animation • Selecting content • Navigating

2.4. Trace Analysis Tools

In this section, we survey existing trace analysis tools. These tools have been developed to help analysts accomplish this process by providing a collection of trace abstraction techniques in a graphical user interface mode and providing a set of features that visualizes the traces and enables interactions with them.

2.4.1. LTTng and LTTV

Desnoyers and Dagenais introduced LTTng (Linux Trace Toolkit Next Generation) as a tracer to extract information from the Linux kernel, user space libraries and from programs by running a recompiled instrumented version of the kernel [10]. It is possible to study the generated trace by using LTTV (Linux Trace Toolkit Viewer), which visualizes the trace and provides a number of views that help developers analyzing the trace [10].

LTTng along with LTTV can be used to reverse engineer programs, libraries, drivers, and even the operating system. It is also useful when used to reason about multi-threaded and multi-process systems running on single or multi-core processors [10].

Figure 4 shows the LTTV main window which is displayed when starting the application [11]. Three main views appear in this window [11]: The Statistic View, to the top, displays statistics about the trace (like the total number of events and overall CPU time), the events' types, the processes, and the CPU (the total number of events executed in each CPU, and the total amount of time consumed at each CPU). The Control Flow View, in the middle, provides an overall view of the trace, which helps developers to detect patterns that are recognized as lines with similar lengths and colors. When detecting a pattern, the user can zoom in to dig deeper into the sequence

of events participating in this pattern. The Detailed Event List View, at the bottom, which displays the list of events related to each process, like entry or exit events.

To deal with large execution traces, LTTV implements optimized algorithms that enable random access to large traces [12]. It also provides a number of visualization techniques like scrolling, zooming, and filtering using logical expressions.

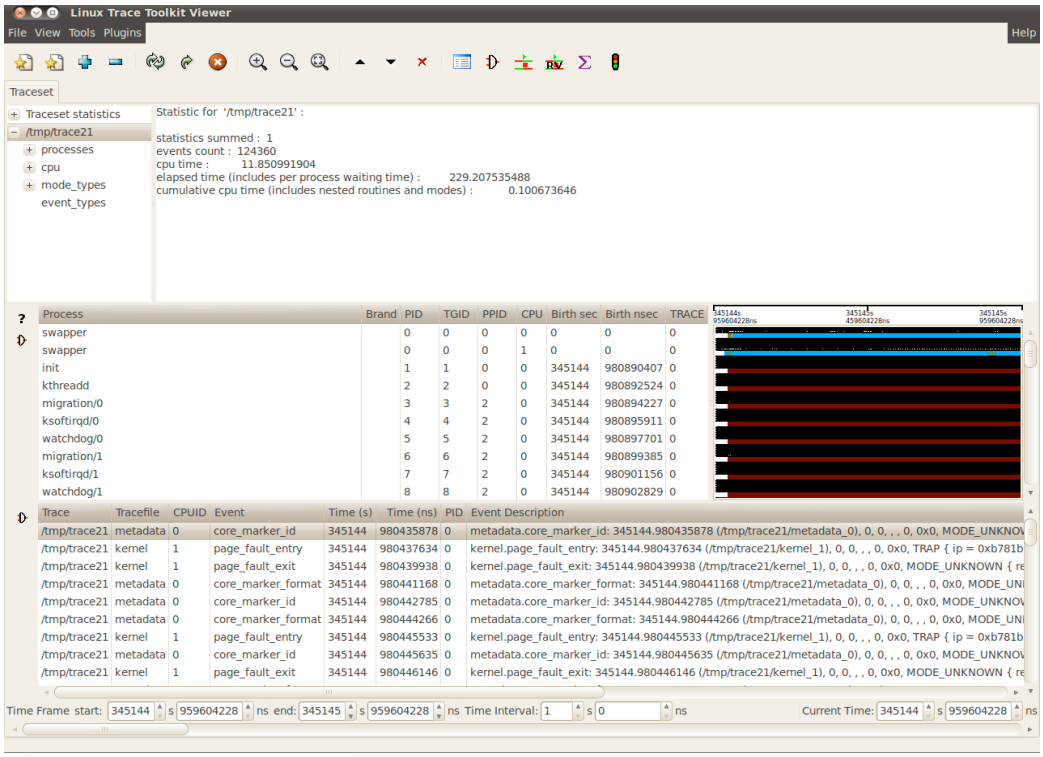

Figure 4. LTTV Main Window

2.4.2. Intel VTune

Intel VTune Performance Analyzer is used to profile applications running on Intel-based systems to help analyzing software performance bottlenecks within the Eclipse framework[1].

Using VTune, traces can be generated in two ways: Sampling and profiling[1]. Sampling is accomplished by interrupting the processor at regular intervals and

[1] http://lttng.org/tracingwiki/index.php/Intel_VTune

collecting samples of instruction addresses[1]. While profiling is accomplished by instrumenting the binary code, and is used to show the program flow and the critical path, which is the most time consuming call sequence[1]. This is accomplished by using the call graph to count the number of calls and the amount of time spent in each function[1].

VTune provides a number of views like process view, thread view, and function view[1]. The process view provides the ability to select a process of the current processes within the system[2]. The thread view shows the execution of the current threads and what the threads are doing[3]. The Function View shows the call graph with performance details related to each function[2].

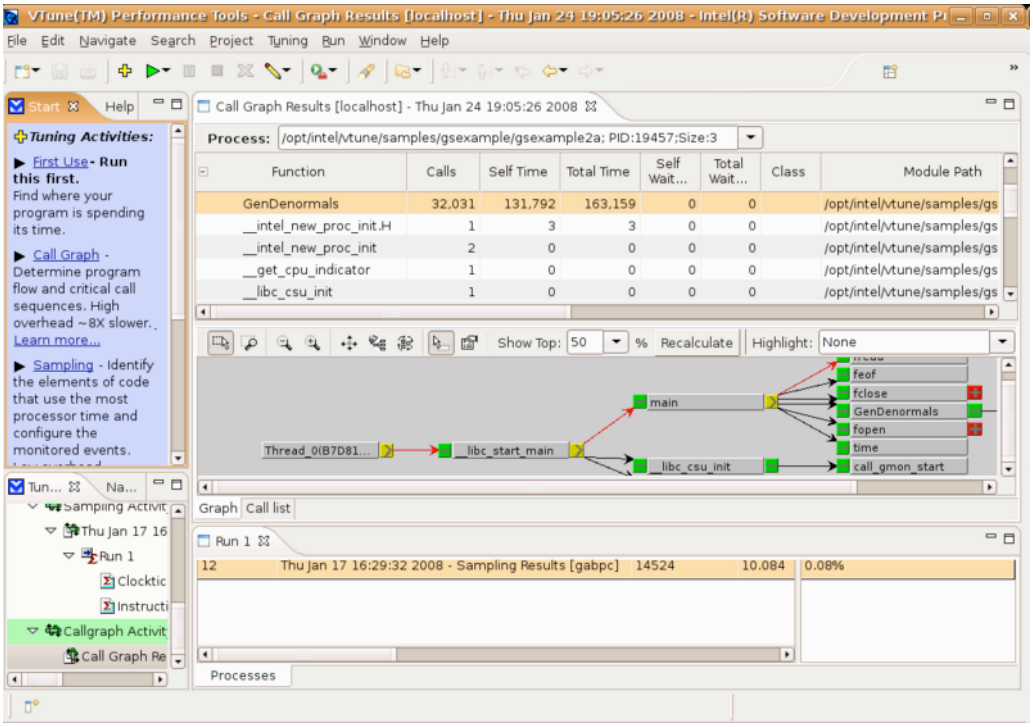

Figure 5. Intel VTune Performance Analyzer (from [46])

[1] http://lttng.org/tracingwiki/index.php/Intel_VTune
[2] http://software.intel.com/en-us/articles/optimizing-for-hyper-threading-technology-using-the-intel-vtunet-performance-analyzer/
[3] http://software.intel.com/en-us/intel-vtune/

To deal with size explosion problem, VTune implements both time-based sampling and event-based sampling techniques (see Section 2.2.)[1]. It also provides a number of visualization techniques like scrolling, zooming, highlighting, and user-guided filtering by process id., CPU number, etc.

2.4.3. SystemTap

SystemTap is a Linux debugger that provides a command line interface and a scripting language to dynamically instrument the Linux kernel and user space programs to analyze the resulting traces[2].

Events are traced by writing a simple SystemTap script with some print statements [32]. For example, the following script is used to trace the open system-call [32]:

```
# cat strace-open.stp
probe syscall.open
{
        printf ("%s(%d) open (%s)\n", execname(), pid(), argstr)
}
```

A sample output line would be:

```
vmware-guestd(2206) open ("/etc/redhat-release", O_RDO)
```

To deal with size explosion problem, SystemTap allows collected data to be filtered, aggregated, transformed, and summarized [32]. This is accomplished through a set of constructs such as if statements, while and for loops, Boolean and mathematical operations, variables, arrays, and functions [32].

[1] http://lttng.org/tracingwiki/index.php/Intel_VTune
[2] http://sourceware.org/systemtap/

2.4.4. WindRiver Workbench

WindRiver Linux is a Linux kernel tracer that uses the LTTng framework as a data provider [33]. It provides a set of tools and views for software development and debugging [33] such as the system viewer (used to display the trace in different ways), the event graph (used to display the thread events), the event table (used to display events as rows of information ordered by their timestamps), and the memory usage graph for displaying memory allocation and deallocation operations. In addition to these views, the tool provides custom filtering, highlighting and selection, and multiple linked views to deal with size explosion problem.

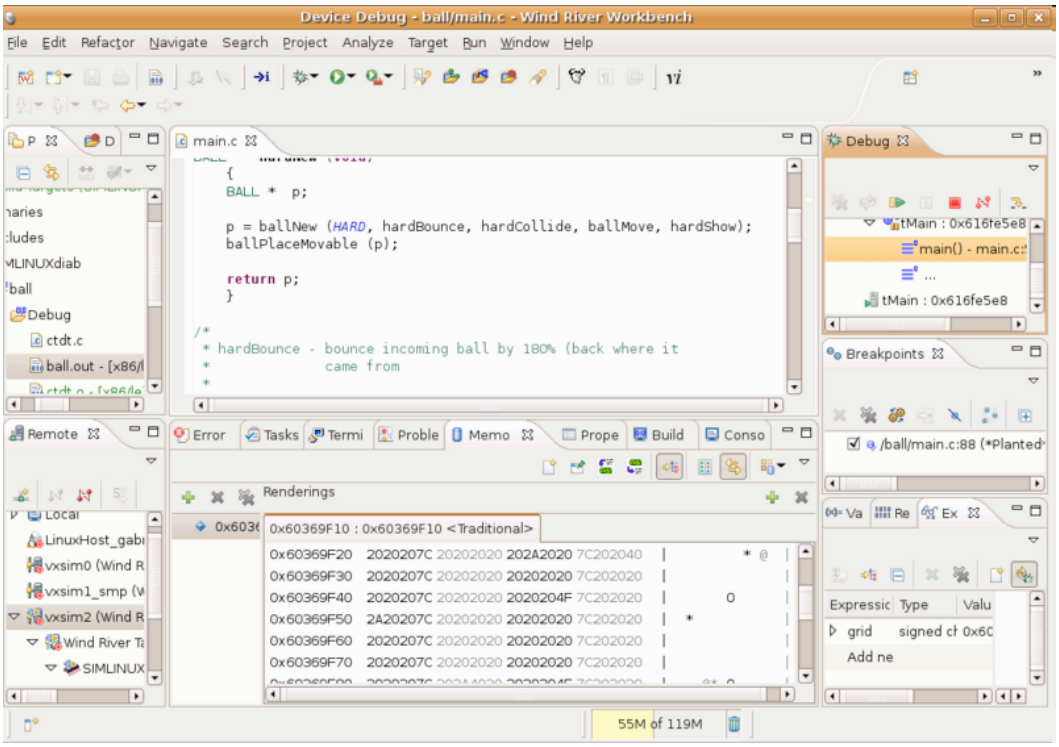

Figure 6. WindRiver Workbench (from [40])

2.4.5. Zealcore System Debugger

Zealcore System Debugger is a debugging software tool used to visualize the logged information through a number of browsers [34].

- The Timeline Browser: It is used to display time distance between events.

- The Gantt Chart: It is used to display the scheduling of threads.
- The Sequence Diagram: It shows processes or objects and messages passed between them.
- The State Diagram: It shows for a given actor the model of its recorded behaviour.
- The Plot View: It is used to draw results of searches as charts.

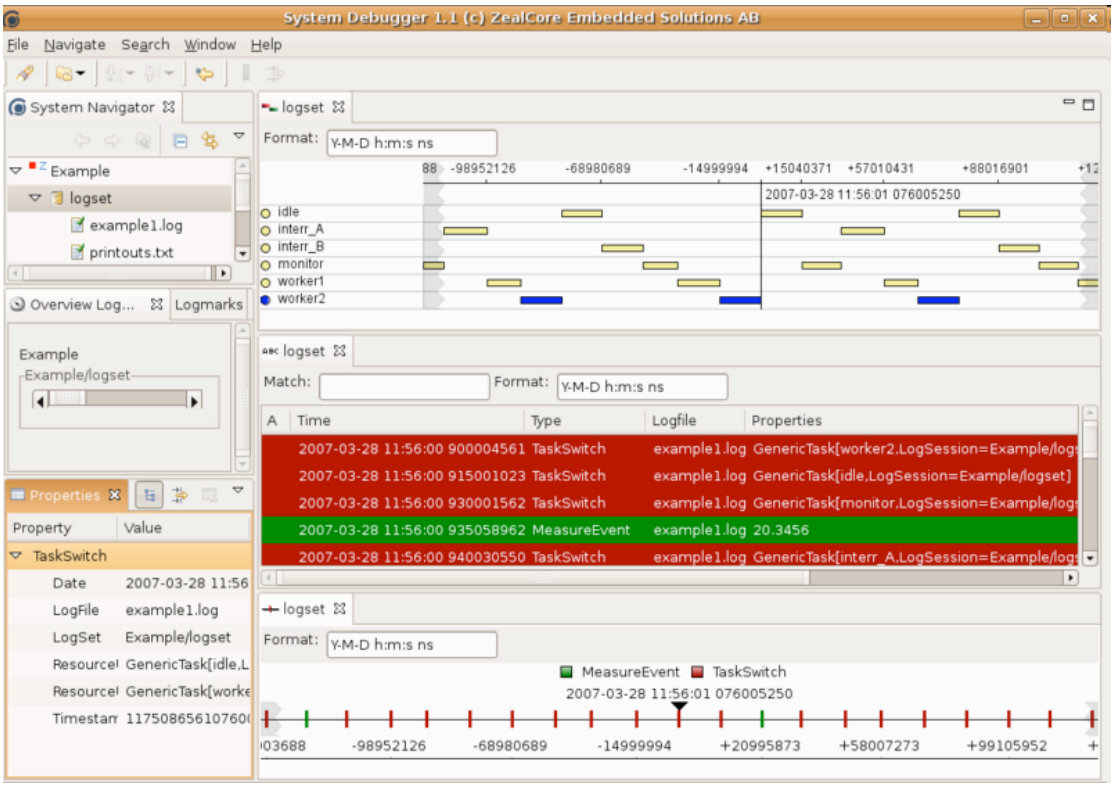

Figure 7. Zeaclore System Debugger (from [35])

To deal with size explosion problem, Zeaclore provides a number of techniques such as searching and filtering, highlighting and selection, zooming and scrolling, and multiple linked views [34].

2.5. Summary

Within this chapter, we showed that many abstraction techniques have been introduced to help developers analyze execution traces. This is because traces tend to

be difficult to work with due to their extraordinary size. This is even more complicated in the case of system call trace due to the presence of sequences resulting from the low-level nature of system calls and the large number of low-level events such as memory management events and hardware/software interrupts.

Though the techniques presented in this chapter aim to reduce a trace size by detecting repeated sequences, removing low-level implementation details, or visualizing the trace, they either operate on routine call traces or depend greatly on specific visualization schemes. None of them focuses on extracting high-level representation from system call traces, which is the objective of this thesis.

Table 2 below summarizes trace abstraction tools and the techniques used by each of them.

Table 2. Trace Abstraction Tools

Abstraction Tool	Multiple Linked Views	Implemented Techniques
LTTng and LTTV	YES	Scrolling, zooming, and filtering
Intel VTune	YES	Scrolling, zooming, highlighting, and filtering
SystemTap	NO	Filtering, aggregating, Transforming, and summarizing
WindRiver Workbench	YES	Filtering, highlighting and selection
Zealcore System Debugger	YES	Searching and filtering, highlighting and selection, zooming and scrolling

Chapter 3. Abstracting System Call Traces

3.1. Approach

As shown in Figure 8, our approach takes a system call trace and applies to it the abstraction process to generate an abstracted version. The abstraction process relies on the Linux pattern library that we have developed to characterize the main operations of the Linux kernel.

To build the pattern library, we have studied the Linux kernel and its system calls mechanism. We also executed a number of applications with different operations and examined the generated traces to understand how the kernel functions. The tracer used in this thesis is LTTng (Linux Tracing Toolkit – new generation) [10]. In the subsequent sections, we elaborate on each step of the abstraction process in more detail.

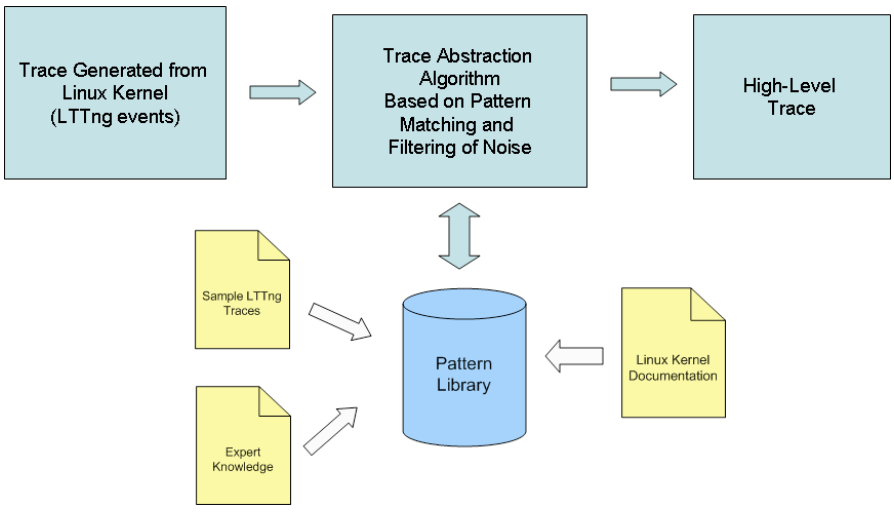

Figure 8. Abstracting system call traces using knowledge-based approach

3.2. The LTTng Instrumentation Tool

LTTng is a tracer developed to extract information from the Linux kernel, user space libraries and from programs by running recompiled instrumented version of the kernel [10].

In our research, we installed LTTng 2.6 on Ubuntu 10 operating system with the kernel version being 2.6.34. It should be mentioned that traces can be generated by directly running LTTng through command line, or by using LTTV through its GUI. We used LTTV approach due to the simplicity offered by the GUI.

LTTV is started by executing the following command in the system's shell:

```
> sudo lttv-gui
```

When the command is executed, the LTTV window appears, from which we choose the LTTng control option to get the following window:

Figure 9. LTTng Control Window

The window provides a number of options that help controlling the tracer behaviour, such as: starting or stopping the tracer, enabling or disabling the probes in the kernel, setting the trace directory and name, and other helpful options.

Once everything is set properly, we can start the tracer, execute our systems, and stop the tracer to get the traces, which are converted into text format through the following command:

```
lttv -m textDump -t path-to-trace-directory -o

path-to-output-text-file
```

An LTTng trace contains information related to the process being executed such as trace file, event name, time in seconds, time in nano seconds, trace file path, process ID, process name, parent ID, process group ID, execution mode, and other parameters related to the event being executed. However this information could vary depending on the selected trace points, hence, they might differ from one version to another and from one testing platform to another. In our research we generate traces with the following format:

```
TraceFile.Event Time(s).Time(ns) (Path_To_Trace_File),
PID, PGID, ProcessName, PPID, MODE {PARAMS}
```

An example of a trace using the above format would be:

```
kernel.syscall_entry: 442192.435342606
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id = 5
[sys_open+0x0/0x40] }
```

```
fs.open: 442192.435348299 (/tmp/trace10/fs_1), 22438,
22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3,
filename = "output.txt" }

kernel.syscall_exit: 442192.435348407
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, USER_MODE { ret = 3 }

kernel.syscall_entry: 442192.435350985
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id = 4
[sys_write+0x0/0xc0] }

fs.write: 442192.435351307 (/tmp/trace10/fs_1), 22438,
22438, ./Files, , 29184, 0x0, SYSCALL { count = 72, fd
= 3 }

kernel.syscall_exit: 442192.435351415
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, USER_MODE { ret = 72 }

kernel.syscall_entry: 442192.435351522
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id = 6
[sys_close+0x0/0x100] }

fs.close: 442192.435351629 (/tmp/trace10/fs_1), 22438,
22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3 }

kernel.syscall_exit: 442192.435351844
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, USER_MODE { ret = 0 }
```

This trace represents the system calls executed when a file is opened, data written to it, and then the file is closed.

It is important to note that the number attached to the trace file indicates the CPU that executes the process, which is CPU 1 in this example, and this is especially useful in the case of multi-core systems, since we want to be able to identify which core executes a particular statement.

3.3. Linux System Call Mechanism

The Linux Kernel acts as an intermediary layer that stands between the software system and the hardware, and it controls the interaction between them [47]. It is designed as a monolithic kernel where all of its functionality, including file management, memory management, device drivers, etc., is implemented in the kernel space [47]. The monolithic design is followed due to the huge gain in performance, when compared to other kernel's architectures such as the microkernel design.

Having the kernel standing as an intermediary layer between the software system and the hardware has the advantage of making it easier for developers to write codes that are free from complicated low-level hardware interaction events which also increase software portability [8]. In addition to that, it makes the system more secure as requests can be checked by the kernel before really executing them [8].

Since a process interacts with the hardware only through the kernel, there had to be some technique that allows such an interaction, and this is where the system call mechanism becomes handy. The mechanism refers to the set of functions that is provided by the kernel through which a process can send a request to the hardware by invoking the appropriate system call as any normal function [47].

As a result, the Linux system can be seen as system divided into a number of layers that are separated by well defined interfaces as shown in Figure 10.

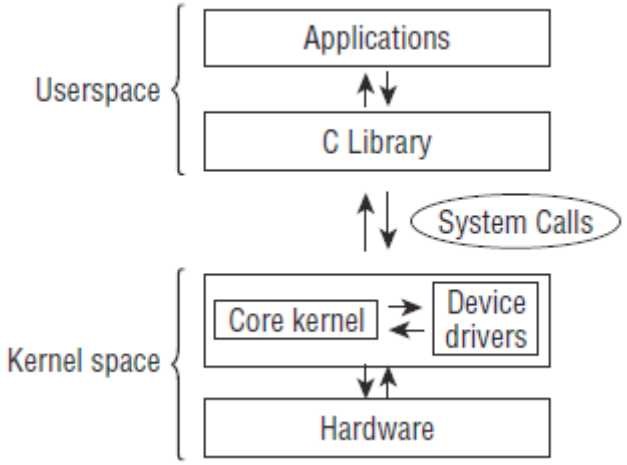

Figure 10. The Linux System (from [47])

3.4. Pattern Library for Linux System Calls

Our main contribution in this thesis is centered around detecting and categorizing patterns of events into a library that can be used to abstract the kernel-space traces and turn them into a more compacted and readable form while preserving their main information.

In Linux, “everything is a file" [27], and this is where we started our work: Detecting patterns for file management operations. Also, as Linux is widely used in networking, we studied socket management patterns. However, files and sockets are both managed through processes, and this is why we also focused on process management patterns. In order to remove noise, we had to work on some memory management, page-faults, and other noise patterns.

In our work, patterns are described as state machines [13, 41] composed of lists of events (transitions) and system modes (states). Events conform to the system calls and

other events that appear in the trace, while states conform to the modes of execution in the trace (USER_MODE_X, SYSCALL_X, etc. where X corresponds to a certain event).

3.4.1. File Management Patterns

In our research, we focused on the most recurrent file management operations including: Open, Read, Write, Seek, Close, Access, File Control, Read Link, Stat, File Duplicate, File Truncate, Device Control, and Poll. Each of these operations and the corresponding state machine that we have developed to characterize how this operation is executed by the Linux kernel is described in what follows:

File Open: It is the action of opening a file from the file system [9], and it is accomplished by entering the sys_open system call, executing the open function in the file system with the appropriate parameters to open the file, and finally exiting the sys_open system call. Figure 11 shows the state machine that models the File Open pattern.

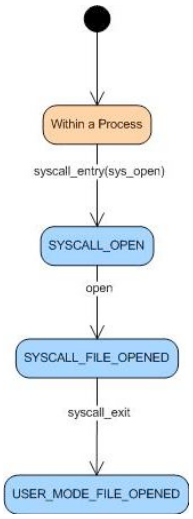

Figure 11. File Open pattern

File Read: It is the action of reading a number of bytes from an opened file [9], and it is accomplished by entering the sys_read system call, executing the read function with the appropriate parameters to read data from the opened file, and finally exiting the sys_read system call. Figure 12 models the File Read pattern.

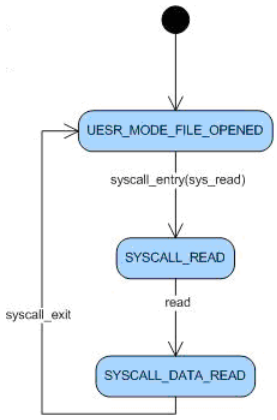

Figure 12. File Read pattern

File Write: It is the action of writing data to an opened file [9], and it is accomplished by entering the sys_write system call, executing the write function with the appropriate parameters to write data to the opened file, and finally exiting the sys_write system call. Figure 13 summarizes the File Write pattern.

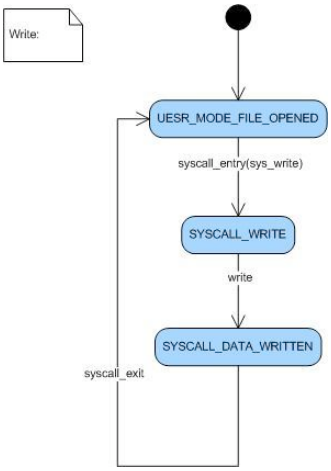

Figure 13. File Write pattern

File Seek: It is the action of changing the position of the file pointer of an opened file [9], and it is accomplished by entering the sys_lseek or sys_llseek system call, executing the lseek or llseek function with the appropriate parameters to change the position of the pointer within the opened file, and finally exiting the system call. Figure 14 summarizes the File Seek pattern.

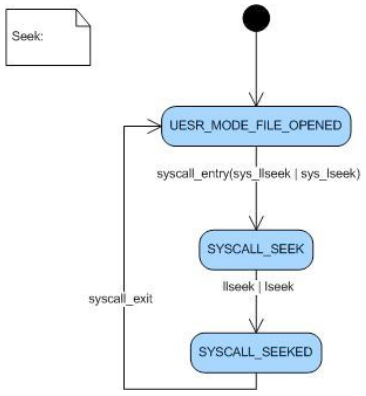

Figure 14. File Seek pattern

File Close: It is the action of closing an opened file [9], and it is accomplished by entering the sys_close system call, executing the close function with the appropriate parameters to close the file, and finally exiting the sys_close system call. Figure 15 summarizes the File Close pattern.

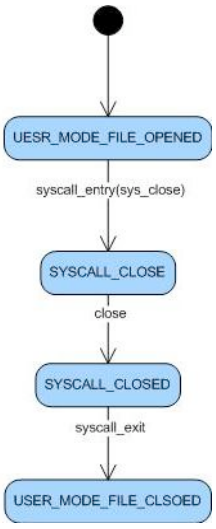

Figure 15. File Close pattern

File Access: It is the action of checking access permissions of a file [9], and it is accomplished by entering the sys_access system call, and exiting the sys_access system call. Figure 16 summarizes the File Access pattern.

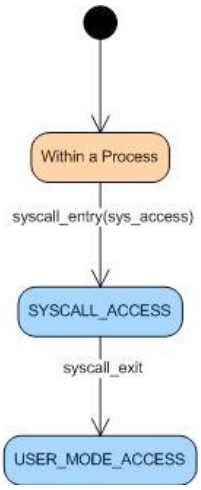

Figure 16. File Access pattern

File Control: It is the action of controlling an opened file descriptor by performing one of the following operations [8]: duplicating file descriptor, handling record locks, notification change for file and directory [26], and many others. It is accomplished by entering the sys_fcntl or sys_fcntl64 system call, and exiting the system call. Figure 17 summarizes the File Control pattern.

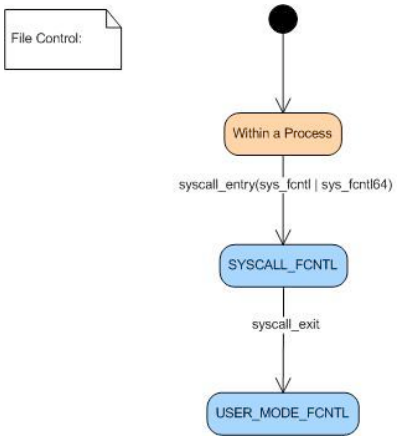

Figure 17. File Control pattern

Read Link: It is the action of reading the contents of a symbolic link [9], and it is accomplished by entering the sys_readlink system call, and exiting the system call. Figure 18 summarizes the Read Link pattern.

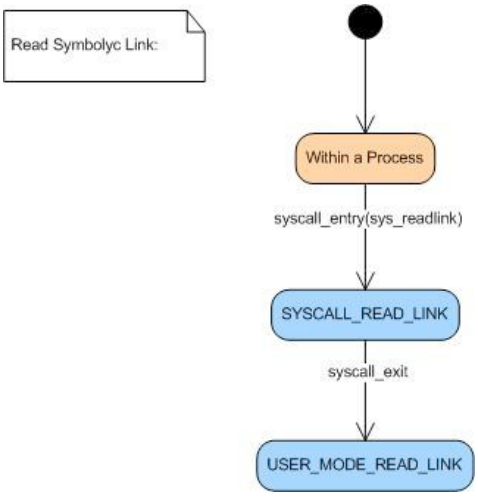

Figure 18. Read Link pattern

File Stat: It is the action of requiring information of a file such as user ID, group ID, total size, etc [9], and it is accomplished by entering any of the stat system calls family like sys_fstat64, sys_fstat, sys_lstat, and sys_stat, and exiting the entered system call. Figure 19 summarizes the File Stat pattern.

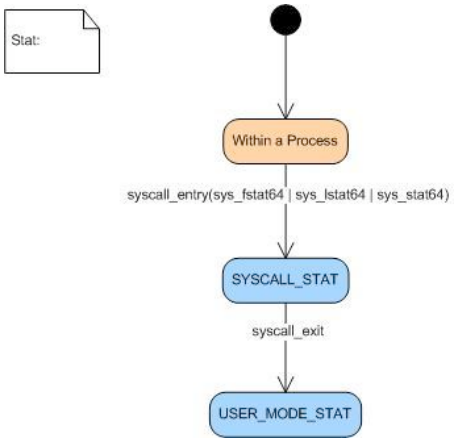

Figure 19. File Stat pattern

File Duplicate: It is the action of duplicating a file descriptor [9], and it is accomplished by entering any of the dup system calls family like sys_dup and sys_dup2, and exiting the entered system call. Figure 20 summarizes the File Duplicate pattern.

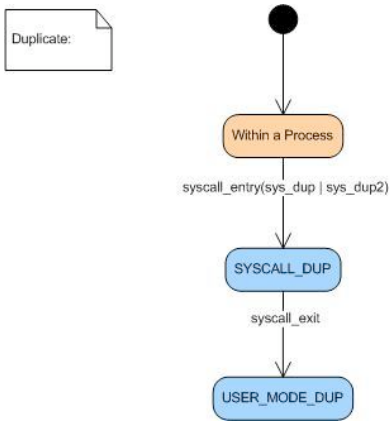

Figure 20. File Duplicate Pattern

File Truncate: It is the action of setting the file size to a specific size [9], and it is accomplished by entering sys_ftruncate system call and exiting it. Figure 21 summarizes the File Truncate pattern.

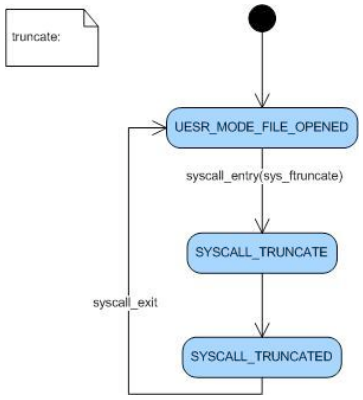

Figure 21. File Truncate Pattern

Device Control: It is the action of controlling devices [9], and it is accomplished by entering sys_ioctl system call, executing ioctl and exiting the entered system call. Figure 22 summarizes the File Control pattern.

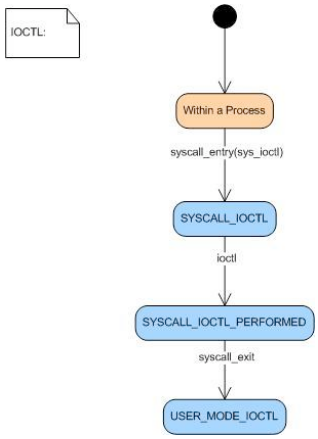

Figure 22. File Control Pattern

Poll: It is the action of waiting on a file descriptor to perform some I/O operation [9], and it is accomplished by entering sys_poll system call, executing pollfd and exiting the entered system call. Figure 23 summarizes the Poll pattern.

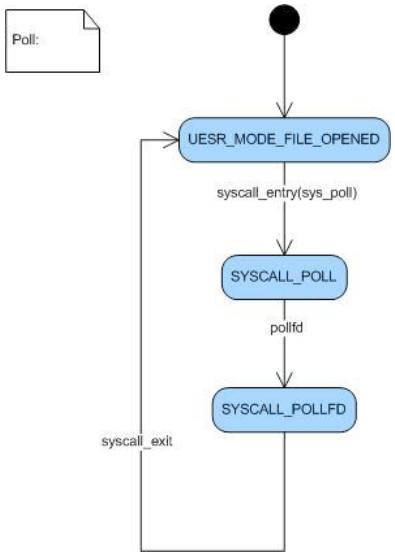

Figure 23. Poll Pattern

We can put the operations that operate on an opened file descriptor into a higher level of abstraction by saying that a file management process is basically a collection of these patterns starting from the File Open pattern, performing any number of readings, writings, etc. Figure 24 summarizes File Management aggregate pattern.

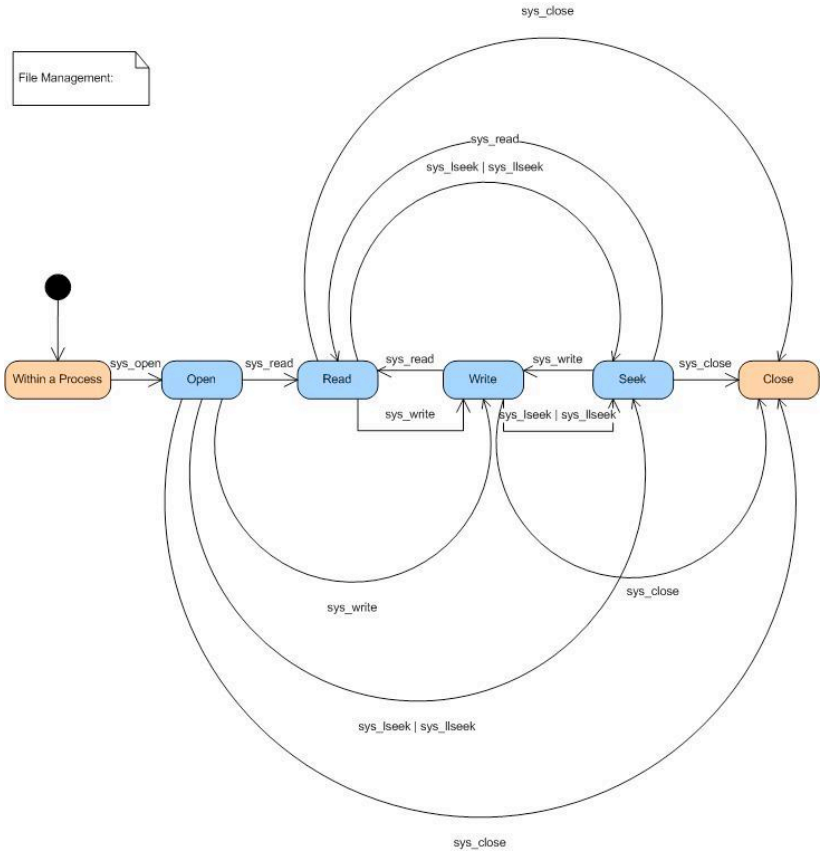

Figure 24. File Management aggregate pattern

3.4.2. Socket Management Patterns

Networking in Linux is accomplished through the concept of a socket [15], which has the following operations: Create, Bind, Connect, Listen, Accept, Send, Receive, and Close.

It is important to note that there are many types of sockets, but we only cover two of them in our work, TCP and UDP sockets [15]. However, the basic operations are the same for both types, except that when using an UDP connection, sockets do not need to connect, listen, or accept connections.

The operations Socket Create, Socket Bind, Socket Connect, Socket Listen, and Socket Accept have similar patterns where each of them is accomplished by entering the sys_socketcall system call [9], executing socket_call with the call type set to 1, 2, 3, 4, or 5, then executing socket_x (where x could be create [8], bind [8], connect [8],

listen [8], and accept [8]) with the appropriate parameters, and finally exiting the sys_socketcall system call. These patterns are shown in Figures 25 through 29.

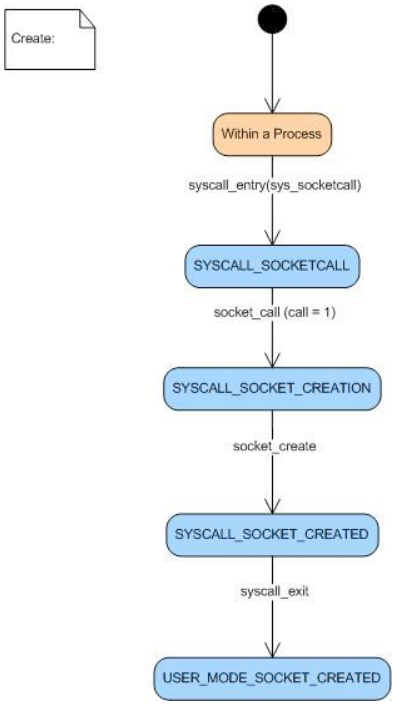

Figure 25. Socket Create pattern

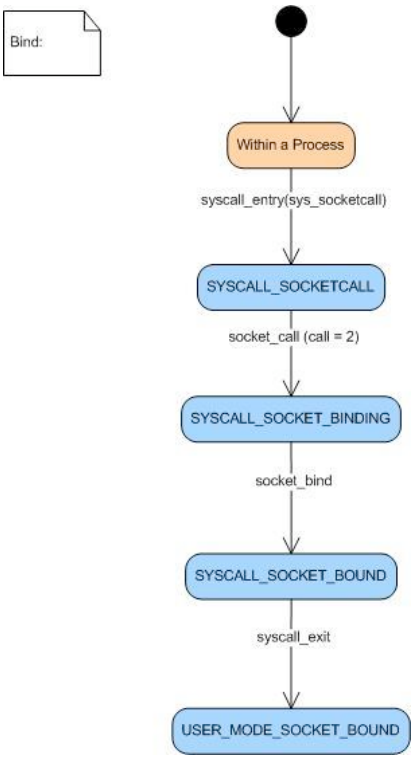

Figure 26. Socket Bind pattern

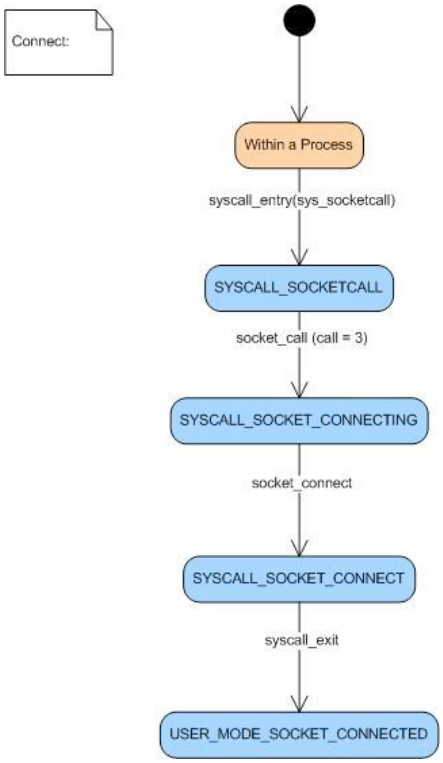

Figure 27. Socket Connect pattern

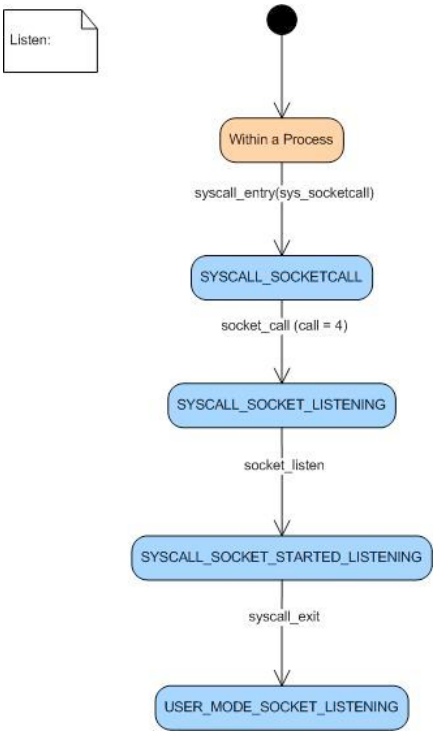

Figure 28. Socket Listen pattern

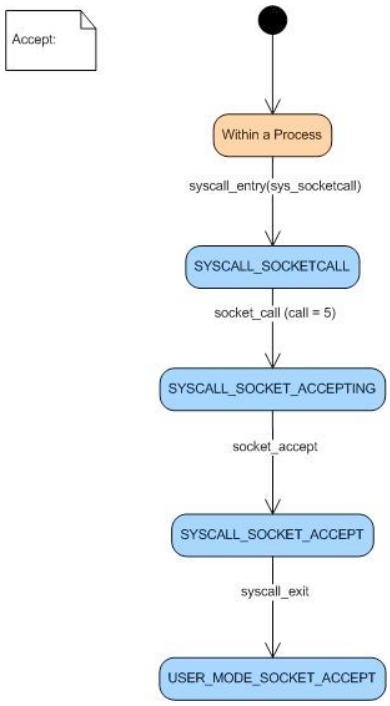

Figure 29. Socket Accept pattern

Socket Send and Socket Receive have similar patterns that present the action of sending or receiving data through opened sockets. This is accomplished by entering the sys_socketcall system call [8], executing socket_call with the call type set to 9 (send [8]), 11 (send to [8], which is used in UDP sockets), 10 (receive [8]), or 12 (receive from [8], which is used in UDP sockets), then executing a number of low-level operations, before finally exiting the sys_socketcall system call. Figure 30 summarizes the Socket Send pattern.

Socket Close pattern is identical to the File Close pattern.

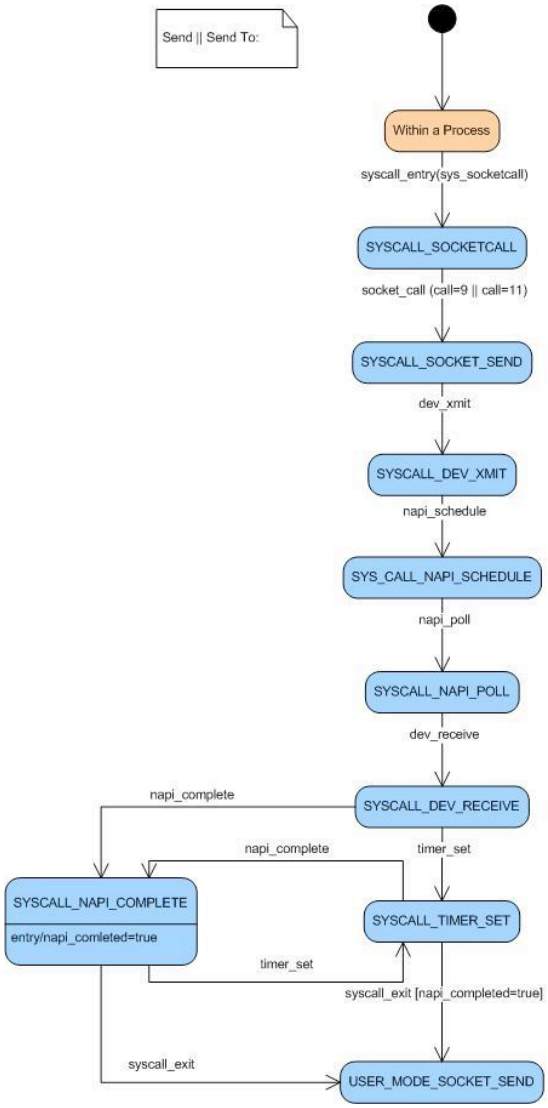

Figure 30. Socket Send pattern

Using these patterns we were able to detect higher-level abstractions that correspond to Linux networking using TCP or UDP sockets. Networking with TCP sockets [15], for example, involves two different processes: The server process and the client process. The server process requires creating a socket, binding it to some port, listening to that port, accepting a connection from a client socket, sending and receiving data, and finally closing the socket [15], while client process requires creating a socket, connecting to a server socket, sending and receiving data, and closing the socket [15]. Figure 31 summarizes the TCP Sockets pattern.

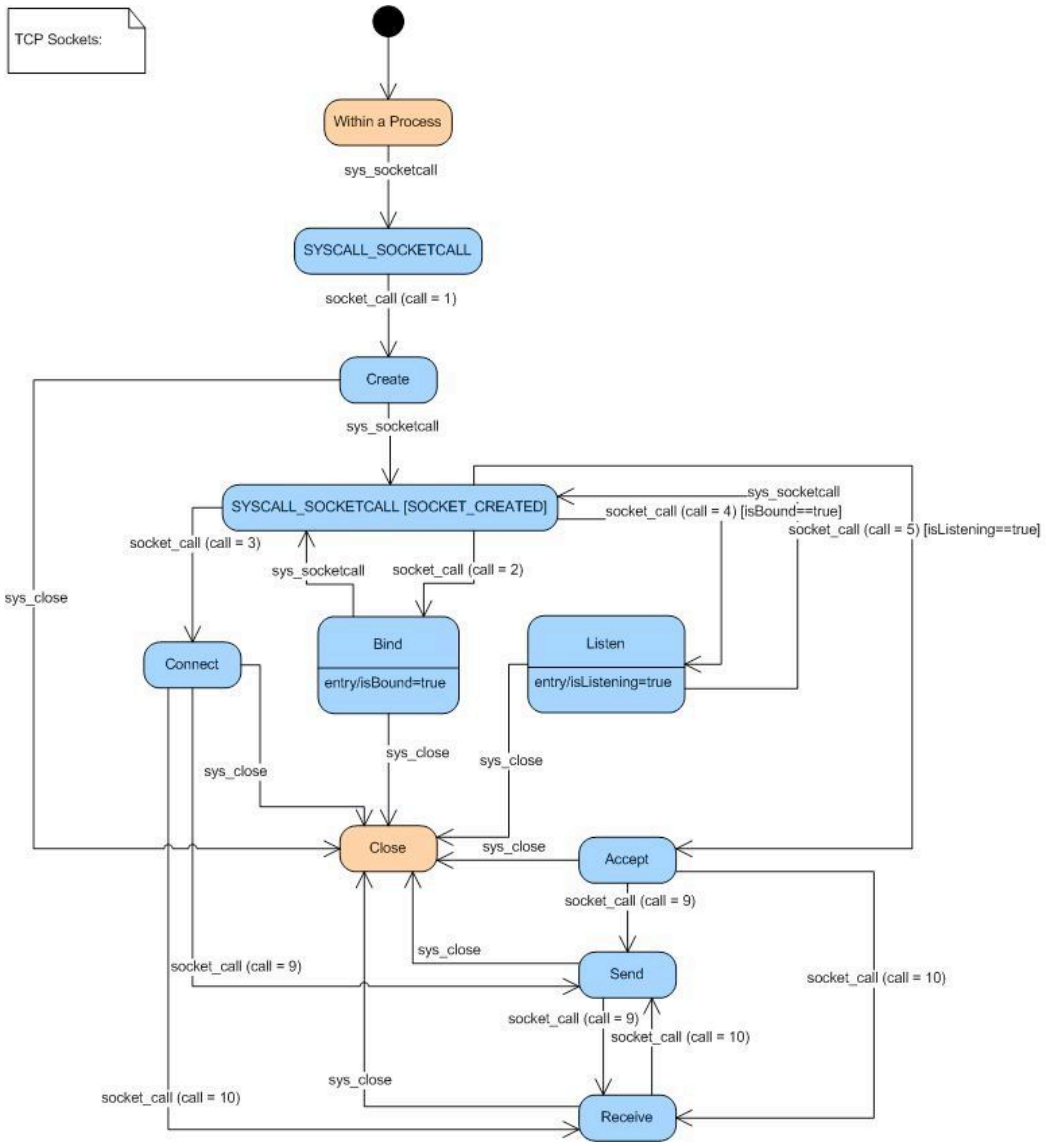

Figure 31. TCP Sockets pattern

With UDP, the server process is run by creating a socket, binding it to a certain address and port, sending and receiving data, and closing the socket [15]. While a client process is similar to the server except that it is not necessary to bind the client process [15]. Figure 32 summarizes UDP Sockets pattern.

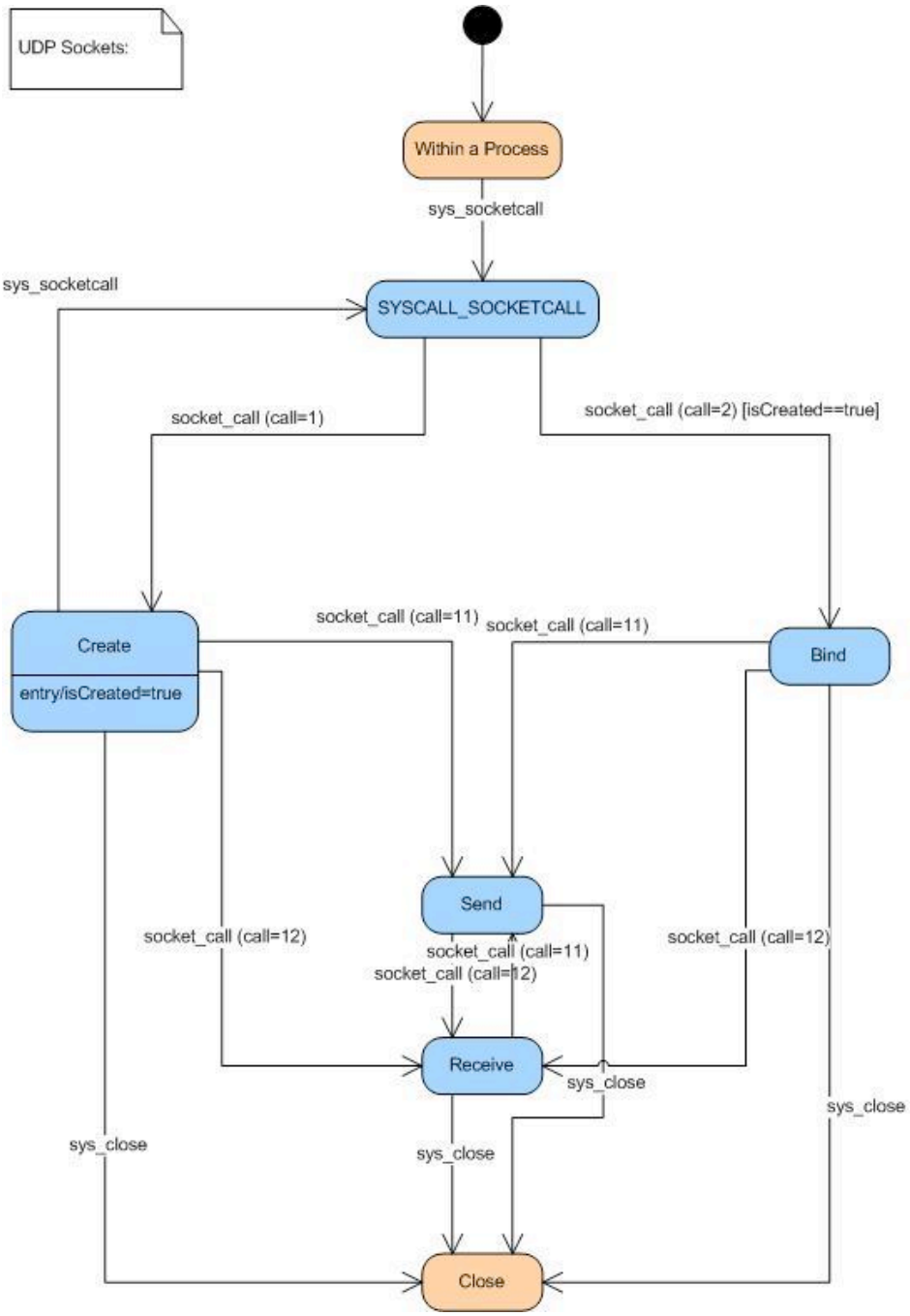

Figure 32. UDP Sockets pattern

3.4.3. Process Management Patterns

In Linux, the "init" process is the parent of all other processes, and it is responsible for managing these processes [8, 14]. This leads to the fact that any operation is executed within a process. As a result, the patterns described in the previous sections appear during process execution, hence it is important to study process management and extract different patterns of process execution. Among many process management patterns, the following patterns are perhaps of most important: Clone, Execute, Get Resource Limit, Get Time of Day, Exit, Get User ID, Get Group ID, Get Process ID, Get Parent Process ID, Set Scheduling Parameters, Get Scheduling Parameters, Get Maximum Scheduling Algorithm Priority, Get Minimum Scheduling Algorithm Priority, Set Scheduling Policy and Parameters, and Unlink.

Process Clone: It is the action of creating a child process with the ability of sharing parts of the execution context with the parent process [9], and it is accomplished by entering the ptregs_clone system call, executing sched_migrate_task, then process_fork, followed by sched_wakeup_new_task, and exiting the entered system call. It is important to note that the clone process is followed by a schedule process in the resulting child process. Figure 33 summarizes the Process Clone pattern.

Process Execute: It is the action of executing a program [9], and is accomplished by entering the ptregs_execve system call, executing sched_try_wakeup, followed by sched_schedule, and finally exiting the entered system call. Figure 34 summarizes the Process Execute pattern.

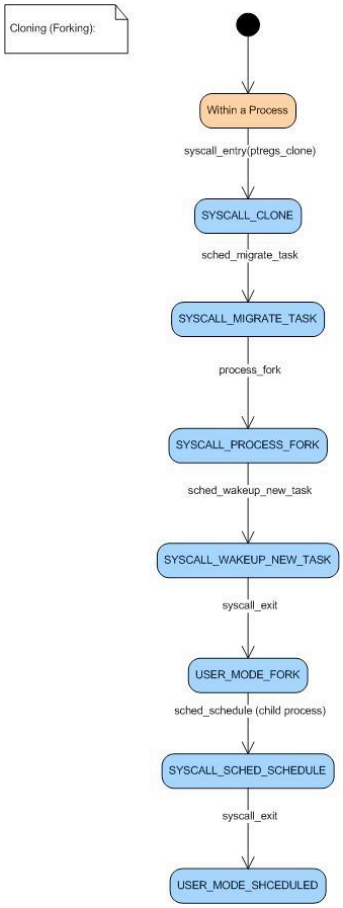

Figure 33. Process Clone pattern

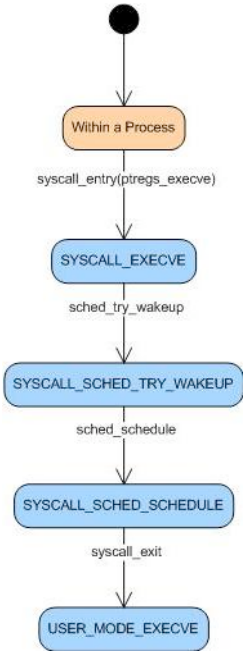

Figure 34. Process Execute pattern

Get Resource Limit: It is the action of getting the soft and hard limit of a resource [9, 26], and is accomplished by entering the sys_getrlimit system call, then exiting that system call. Figure 35 summarizes Get Resources Limit pattern.

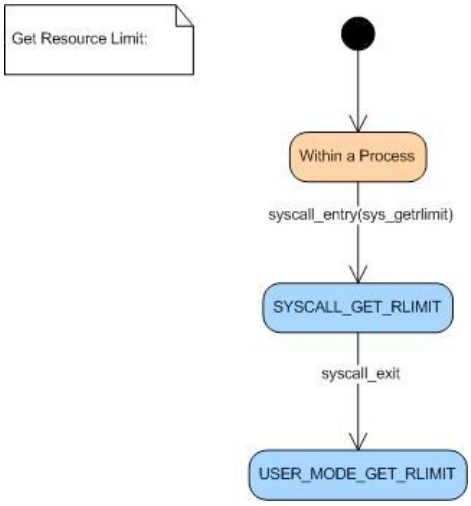

Figure 35. Get Resource Limit pattern

Get Time of Day: It is the action of retrieving the time [9], and is accomplished by entering the sys_gettimeofday system call, then exiting that system call. Figure 36 summarizes Get Time of Day pattern.

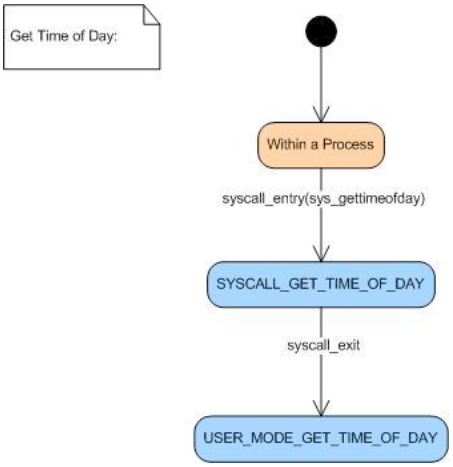

Figure 36. Get Time of Day pattern

Process Exit: It is the action of exiting a process [9] and is accomplished by entering the sys_exit_group system call, executing process_exit, followed by send_signal, and finally sched_try_wakeup. Figure 37 summarizes the Process Exit pattern.

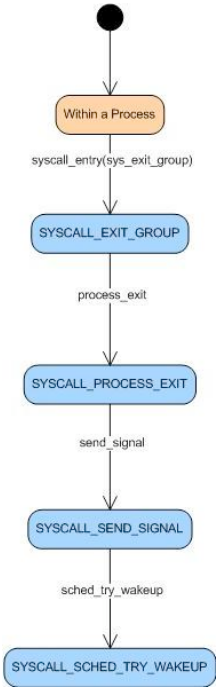

Figure 37. Process Exit pattern

Get User ID: It is the action of getting the user ID of a process [9] and is accomplished by entering the sys_getuid system call, and exiting it. Figure 38 summarizes the Get User ID pattern.

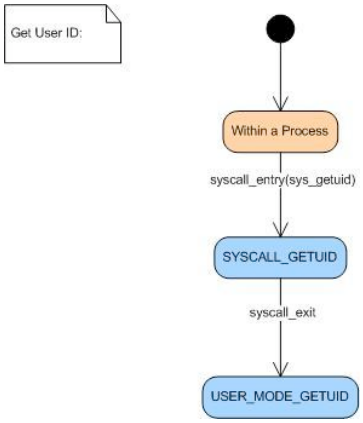

Figure 38. Get User ID Pattern

Get Group ID: It is the action of getting the group ID of a process [9] and is accomplished by entering the sys_getgid system call, and exiting it. Figure 39 summarizes the Get Group ID pattern.

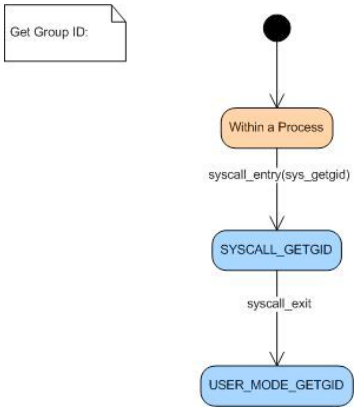

Figure 39. Get Group ID Pattern

Get Process ID: It is the action of getting the process ID [9] and is accomplished by entering the sys_getpid system call, and exiting it. Figure 40 summarizes the Get Process ID pattern.

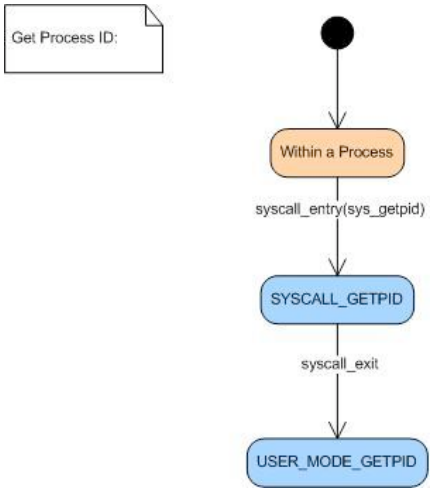

Figure 40. Get Process ID Pattern

Get Parent Process ID: It is the action of getting the parent process ID [9] and is accomplished by entering the sys_getppid system call, and exiting it. Figure 41 summarizes the Get Parent Process ID pattern.

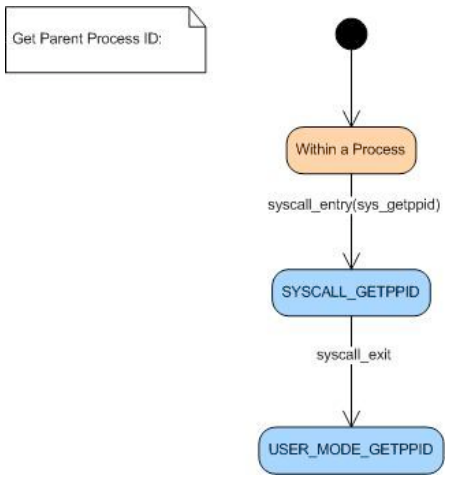

Figure 41. Get Parent Process ID Pattern

Set Scheduling Parameters: It is the action of setting the scheduling parameters for a process [8] and is accomplished by entering the sys_sched_setparam system call, and exiting it. Figure 42 summarizes the Set Scheduling Parameters pattern.

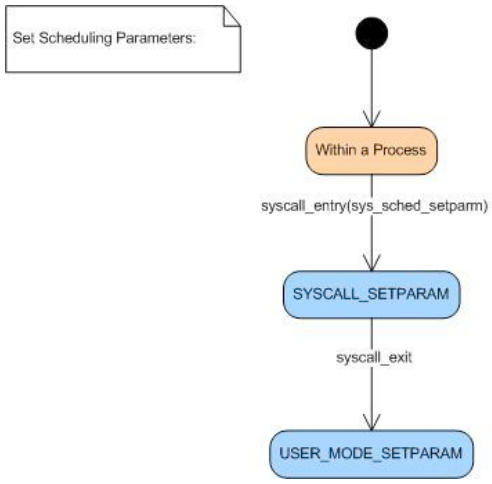

Figure 42. Set Scheduling Parameters Pattern

Get Scheduling Parameters: It is the action of getting the scheduling parameters for a process [8] and is accomplished by entering the sys_sched_getparam system call, and exiting it. Figure 43 summarizes the Get Scheduling Parameters pattern.

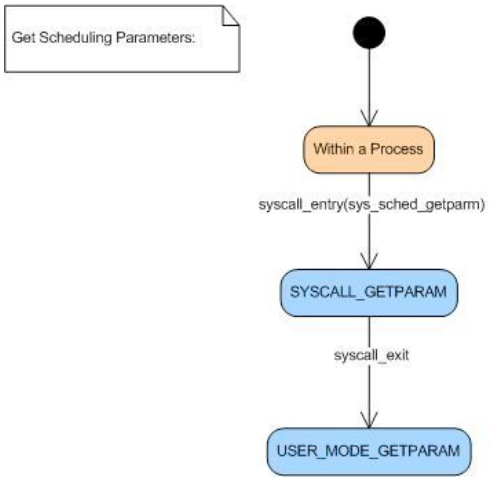

Figure 43. Get Scheduling Parameters Pattern

Get Maximum Scheduling Algorithm Priority: It is the action of getting the maximum scheduling algorithm priority for a process [8] and is accomplished by entering the sys_sched_get_priority_max system call, and exiting it. Figure 44 summarizes the Get Maximum Scheduling Algorithm Priority pattern.

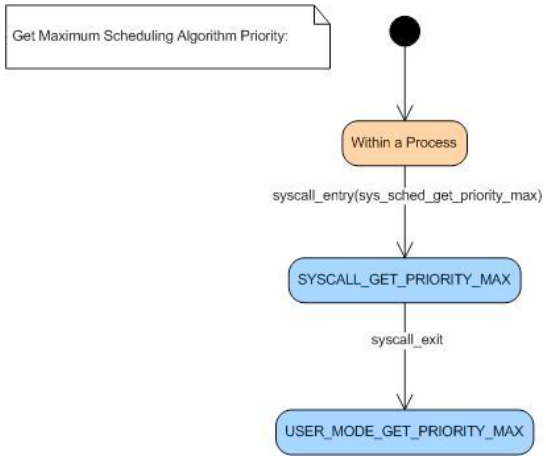

Figure 44. Get Maximum Scheduling Algorithm Priority Pattern

Get Minimum Scheduling Algorithm Priority: It is the action of getting the minimum scheduling algorithm priority for a process [8] and is accomplished by entering the sys_sched_get_priority_min system call, and exiting it. Figure 45 summarizes the Get Minimum Scheduling Algorithm Priority pattern.

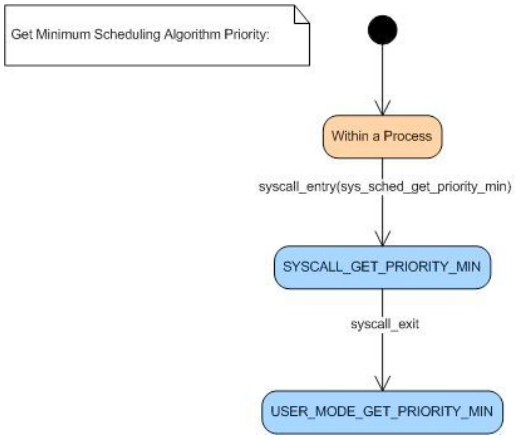

Figure 45. Get Minimum Scheduling Algorithm Priority Pattern

Set Scheduling Policy and Parameters: It is the action of setting the scheduling policy and parameters for a process [8] and is accomplished by entering the sys_sched_setshceduler system call, and exiting it. Figure 46 summarizes the Set Scheduling Policy and Parameters pattern.

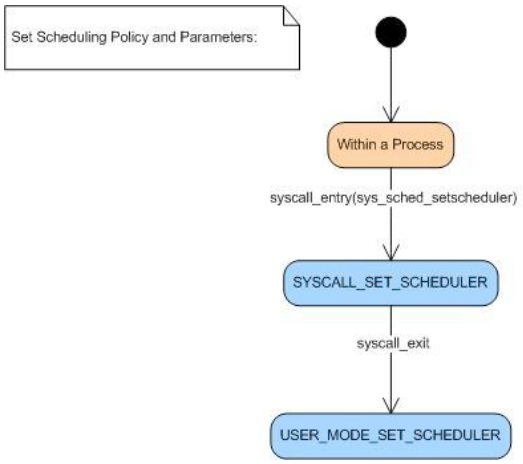

Figure 46. Set Scheduling Policy and Parameters Pattern

Unlink: It is the action of deleting a name from the file system, and the file that is referred by this link if it was the last link [9, 26] and is accomplished by entering the sys_unlink system call, and exiting it. Figure 47 summarizes the Unlink pattern.

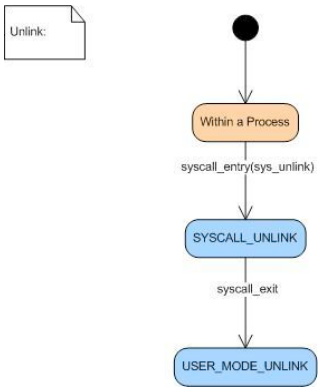

Figure 47. Unlink Pattern

When a process is started within a Linux shell, the following operations are executed in the parent process (which is the shell process itself): Clone and Wait, to create a child process. Then, the Execute operation is executed in the resulting child process. After that, other operations could be executed in the child process depending on the corresponding program, such as: Sockets, Files, Memory Management and Exit. When the parent process is scheduled again, it will be able to execute other operations. Figure 48 summarizes a sample execution of a process within the Linux shell.

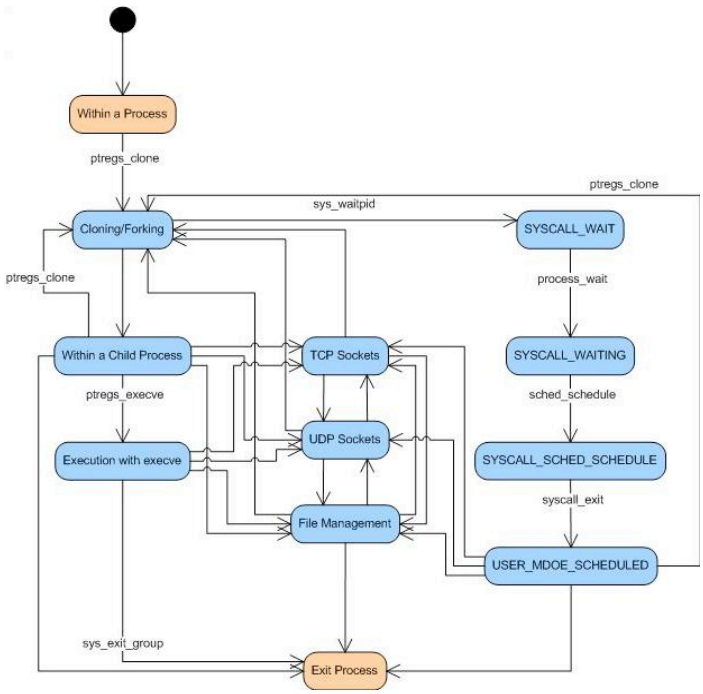

Figure 48. Execution of a process within the Linux shell

3.4.4. Noise Patterns

It is important to note that the aforementioned patterns are usually polluted with what we consider noise patterns, such as memory management patterns, page faults patterns, hardware interrupts patterns, etc. These patterns can appear anywhere in the trace in a non-predictable way, and they do not add valuable information to the system behaviour.

We cover four Memory Management patterns in this thesis, resulting from entering one of the following four system calls: sys_mprotect [9], sys_mmap2 [9], sys_brk [9], sys_set_thread_area [42], and sys_munmap [9] (which happens after sys_mmap2), followed by a number of calls to classic_call_rcu [43] or page_free [44], and finally existing the entered system call. Figure 49 summarizes Memory Management patterns.

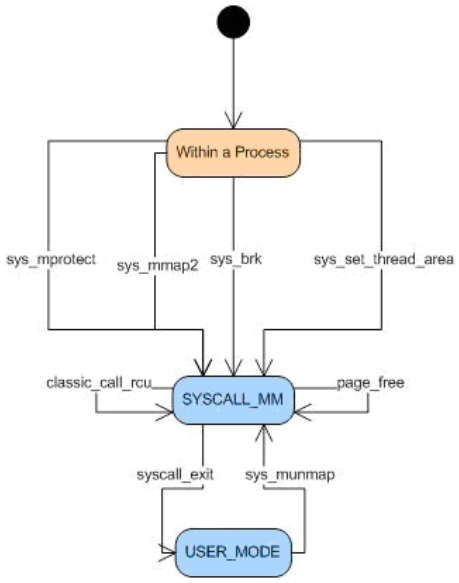

Figure 49. Memory Management patterns

Page Fault [8]: It is detected when executing a page_fault_entry, followed by 0 or more page_alloc [44], and finally executing a page_fault_exit. Figure 50 summarizes Page Fault pattern.

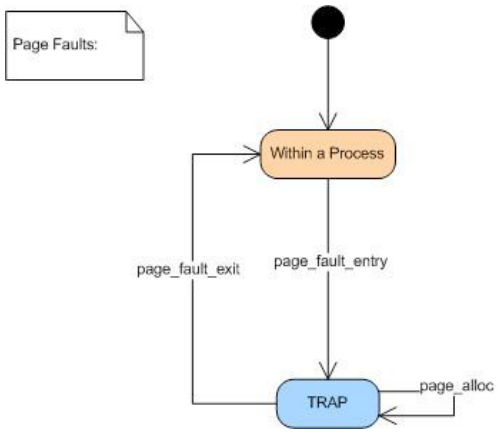

Figure 50. Page Fault pattern

3.5. Trace Abstraction Process

Our trace abstraction algorithm is based on pattern detection and noise removal. The algorithm takes an LTTng trace as an input, and returns an abstracted trace as an output. It starts by parsing the trace from the first line, comparing each event with the events patterns exist in our pattern library until a match is found. Once this happens, the pattern containing the event is shifted from its old state to a newer state waiting for the next event to be read. After that, a new line is read by the algorithm, and the events are compared. When an event causes a pattern to be shifted to a final state, a new high-level construct representing that pattern is created and pushed into a stack of high-level constructs.

When the algorithm has finished processing the entire trace, the patterns of events will be replaced with higher-level constructs ordered in a stack that reflects the system behaviour in a more compact and readable format. Some high-level constructs are marked as noise events. These constructs can be hidden when needed. Also, the resulting constructs could be further abstracted to get a higher-level view of the system. For example, File Open, File Read, File Close constructs could be replaced with File Management construct.

The algorithm is summarized by the following pseudo code:

```
Tokenize the trace
FOR every token within the trace DO
    FOR every pattern in the library DO
            IF pattern can interpret token THEN
                 SHIFT current pattern state
                 IF trace is in final state THEN
                       Generate HIGH LEVEL CONSTRUCT
                 END IF
                 break;
            END IF
   END FOR
   IF token not interpreted THEN
        generate UNKNOWN EVENT
   END IF
END FOR
```

As an example, consider the following trace which consists of 10 events:

```
kernel.syscall_entry: 442192.435342606 (/tmp/trace10/kernel_1), 22438, 22438,
./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id = 5
[sys_open+0x0/0x40] }

fs.open: 442192.435348299 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184,
0x0, SYSCALL { fd = 3, filename = "output.txt" }

kernel.syscall_exit: 442192.435348407 (/tmp/trace10/kernel_1), 22438, 22438,
./Files, , 29184, 0x0, USER_MODE { ret = 3 }

kernel.syscall_entry: 442192.435350985 (/tmp/trace10/kernel_1), 22438, 22438,
./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id = 4
[sys_write+0x0/0xc0] }
```

```
mm.page_alloc: 442192.435351092 (/tmp/trace10/mm_1), 22438, 22438, ./Files, ,
29184, 0x0, SYSCALL { pfn = 79953, order = 0 }

fs.write: 442192.435351307 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184,
0x0, SYSCALL { count = 72, fd = 3 }

kernel.syscall_exit: 442192.435351415 (/tmp/trace10/kernel_1), 22438, 22438,
./Files, , 29184, 0x0, USER_MODE { ret = 72 }

kernel.syscall_entry: 442192.435351522 (/tmp/trace10/kernel_1), 22438, 22438,
./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id = 6
[sys_close+0x0/0x100] }

fs.close: 442192.435351629 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184,
0x0, SYSCALL { fd = 3 }

kernel.syscall_exit: 442192.435351844 (/tmp/trace10/kernel_1), 22438, 22438,
./Files, , 29184, 0x0, USER_MODE { ret = 0 }
```

The first step would be to tokenize the trace, and as a result we get the following sets of tokens (some tokens have been omitted for the purpose of simplicity):

```
[syscall_entry, /tmp/trace10/kernel_1, ./Files, SYSCALL, sys_open+0x0/0x40]

[open, /tmp/trace10/fs_1, ./Files, SYSCALL, fd = 3, filename = "output.txt" ]

[syscall_exit, /tmp/trace10/kernel_1, ./Files, USERMODE, ret=3]

[syscall_entry, /tmp/trace10/kernel_1, ./Files, SYSCALL, sys_write+0x0/0xc0]

[page_alloc, /tmp/trace10/mm_1, ./Files, SYSCALL]

[write, /tmp/trace10/fs_1, ./Files, SYSCALL, count = 72, fd = 3]

[syscall_exit, /tmp/trace10/kernel_1, ./Files, USER_MODE, ret = 72]

[syscall_entry, 442192/tmp/trace10/kernel_1, ./Files, SYSCALL, sys_close+0x0/0x100]

[close, /tmp/trace10/fs_1, SYSCALL, fd = 3]

[syscall_exit, /tmp/trace10/kernel_1, USER_MODE]
```

In the second step, the first token is compared to every pattern in the library until there is a match between the [event name, syscall name] of the token with the [event name, syscall name] of the pattern which is in this case [syscall_entry, sys_open].

When a match happens, i.e. when the current pattern is File Open, the trace state is shifted into SYSCALL_OPEN.

Next, the second set of tokens is read and compared to every pattern in the library, until the algorithm hits a pattern with the current state SYSCALL_OPEN and the next expected event being [open].

When that happens, the current pattern's state is shifted into SYSCALL_FILE_OPENED.

Then, the third set of tokens is read and compared to every pattern in the library, until the algorithm hits a pattern with the current state SYSCALL_FILE_OPENED and the next expected event being [syscall_exit]. When this happens, a high level construct is generated with the following information:

```
File Open:  442192.435348299 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0,
SYSCALL { fd = 3, filename = "output.txt" }
```

After that, the fourth set of tokens is read and compared to every pattern in the library, until the algorithm finds a pattern with the expected event being [syscall_entry, syscall_write]. When that happens, the current pattern's state is shifted to SYSCALL_WRITE.

Then, the fifth set of tokens is read and compared to every pattern in the library, until the algorithm hits a pattern with the expected event being [page_alloc].

A Page Alloc event is generated and marked as a noise event that would not be displayed in the output.

After that, the sixth set of tokens is read and compared to every pattern in the library, until the algorithm hits a pattern with the current state SYSCALL_WRITE and the next expected event being [write]. The current pattern's state is shifted to SYSCALL_DATA_WRITTEN.

Next, the seventh set of tokens is read and compared to every pattern in the library, until the algorithm hits a pattern with the current state SYSCALL_DATA_WRITTEN and the next expected event being [syscall_exit]. A high level construct is generated with the following information:

```
File Write:  fd = 3
```

After that, the eighth set of tokens is read and compared to every pattern in the library, until the algorithm hits a pattern with the expected event being [syscall_entry, syscall_close]. The current pattern's state is shifted to SYSCALL_CLOSE.

Then, the ninth set of tokens is read and compared to every pattern in the library, until the algorithm hits a pattern with the current state SYSCALL_CLOSE and the next expected event being [close]. The current pattern's state is shifted to SYSCALL_ CLOSED.

Finally, the tenth set of tokens is read and compared to every pattern in the library, until the algorithm hits a pattern with the current state SYSCALL_CLOSED and the next expected event being [syscall_exit]. A high level construct is generated with the following information:

```
File or Socket Close:  442192.435351629 (/tmp/trace10/fs_1), 22438, 22438, ./Files, ,
29184, 0x0, SYSCALL { fd = 3 }
```

As a result the initial ten-event trace is replaced with the following three-event high level trace.

```
File Open:  442192.435348299 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0,
SYSCALL { fd = 3, filename = "output.txt" }

File Write: { fd = 3 }

File or Socket Close:  442192.435351629 (/tmp/trace10/fs_1), 22438, 22438, ./Files, ,
29184, 0x0, SYSCALL { fd = 3 }
```

3.6. Summary

In this chapter, we introduced our approach to abstract system call traces generated from the Linux kernel, which is a pattern-based approach where a pattern library for the Linux system calls has been developed and used to abstract traces generated from the kernel. This library includes patterns of the most recurrent operations within the Linux system such as: File Management (Open, Read, Write, Seek, Access, Stat, Control, Read Link, Close), Socket Management (Create, Bind, Connect, Listen, Accept, Send, receive, Close), and Process Management (Execute, Clone, Get Resource Limit, Get Time of Day, Exit), in addition to Noise patterns that can simply be hidden. Finally, we designed an algorithm that abstracts a trace by comparing the events to the patterns defined within the library, and making a decision about replacing a number of events with a higher-level construct and hiding that construct if it corresponds to a utility (noise) construct.

Table 3 shows a list of the system calls that have been studied and their descriptions.

Table 3. List of most important systems calls in our study

Category	System Call	Description
File Management	sys_open	Opening a file from the file system
	sys_read	Reading a number of bytes from an opened file
	sys_write	Writing data to an opened file
	sys_lseek, sys_llseek	Changing the position of the file pointer of an opened file
	sys_close	Closing an opened file
	sys_access	Checking access permissions of a file
	sys_fcntl, sys_fcnyl64	Controlling an opened file descriptor
	sys_readlink	Reading the contents of a symbolic link
	sys_fstat64, sys_fstat, sys_lstat, sys_stat	Requiring information of a file
	sys_dup, sys_dup2	Duplicating a file descriptor
	sys_ftruncate	Setting the file size to a specific size
	sys_ioctl	Controlling devices
	sys_poll	Waiting on a file descriptor to perform some I/O operation
Socket Management	sys_socketcall	This system call is entered when executing one of the following operations on sockets: create, bind, connect, listen, accept, send and receive
Process Management	ptregs_clone	Creating a child process
	ptregs_execve	Executing a program
	sys_getrlimit	Getting the soft and hard limit of a resource
	sys_gettimeofday	Retrieving the time

	sys_exit_group	Exiting a process
	sys_getuid	Getting the user ID of a process
	sys_getgid	Getting the group ID of a process
	sys_getpid	Getting the process ID
	sys_getppid	Getting the parent process ID
	sys_sched_setparam	Setting the scheduling parameters for a process
	sys_sched_getparam	Getting the scheduling parameters for a process
	sys_sched_get_priority_max	Getting the maximum scheduling algorithm priority for a process
	sys_sched_get_priority_min	Getting the minimum scheduling algorithm priority for a process
	sys_sched_setshceduler	Setting the scheduling policy and parameters for a process
	sys_unlink	Deleting a name from the file system, and the file that is referred by this link if it was the last link

Chapter 4. Application

In this chapter we evaluate our approach by applying it to traces generated from five different systems that were run on the Linux kernel, instrumented with LTTng.

The rest of this chapter is organized as follows: In Section 4.1, we introduce the System Call Abstraction Tool that we have developed to implement our techniques. In Section 4.2, we introduce the target systems and discuss their main attributes. In Section 4.3, we describe the process of the generating traces from the target systems. In Section 4.4, we describe the application of our abstraction techniques on the generated traces. Quantitative and qualitative analysis for the results are presented in Section 4.5.

4.1. System Call Abstraction Tool

To test our approach, we had to design and implement a tool that takes as an input a trace generated from LTTng tracer, then applies our algorithm on that trace, and finally outputs the trace in its abstracted form.

The tool was designed following a number of software engineering design principles and patterns. Its architecture was partitioned both horizontally and vertically. As a result, we were able to design a tool that is easy to extend (to add new system calls patterns or even patterns for different calls) and maintain (to modify existing patterns or the design itself).

Horizontal Partitioning:

Horizontal partitioning is performed by defining the main domains of the system, and as a result dividing it into a number of packages, as shown in Figure 51.

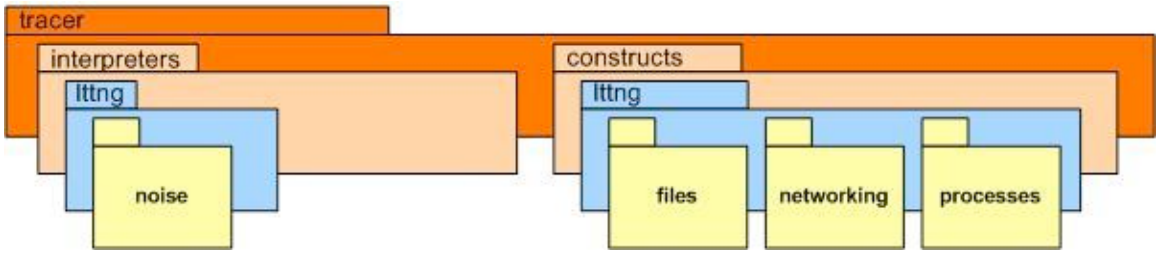

Figure 51. Horizontal Partitioning

As we can see from Figure 51, the system is divided into two main packages: interpreters and constructs. The interpreters package provides interfaces and abstract classes required to add new interpreters that correspond to new patterns. Inside this package, we have the lttng package which is a sample implementation package that provides concrete interpreters like FileInterpreter, SocketInterpreter, etc. Therefore, it is possible to add patterns that are generated from other tools and correspond to different types of calls by simply adding a new package with concrete classes that implement the interfaces provided by the interpreters package. The noise package is a sample implementation package that provides concrete noise interpreters.

The constructs package provides interfaces and abstract classes required to add new high-level constructs that replace the patterns in the abstracted trace. Inside this package the lttng package which is a sample implementation package that provides concrete constructs like HighLevelNoiseConstruct, HighLevelUnknownEventConstruct, etc. Inner packages such as files, networking,

and processes packages are sample implementation packages that provide concrete constructs for file, socket, and process management respectively.

Vertical Partitioning:

Vertical partitioning is applied to divide the system into different layers and define the interfaces between these layers, in such a way that makes it easier to design, implement and maintain each layer independently from the others. Figure 52 shows the vertical partitioning.

- The presentation layer holds the graphical user interface packages.
- The application layer holds the facades and factories required to create appropriate interpreters and communicate with them.
- The business logic layer holds the system packages required to extract high-level constructs using the predefined patterns library.
- The data storage layer is used to store trace files and is controlled by the business logic layer.

The advantage gained from dividing the system into layers can be summarized as the following:

- The dependency flow goes from higher layers to lower layers.
- Layers communicate through interfaces.
- Presentation layer can be developed without affecting any lower layer (multiple GUIs can be provided for the same data).
- Lower layers can provide different implementations without affecting the presentation layer.

Components from different layers can be designed, implemented and maintained independently.

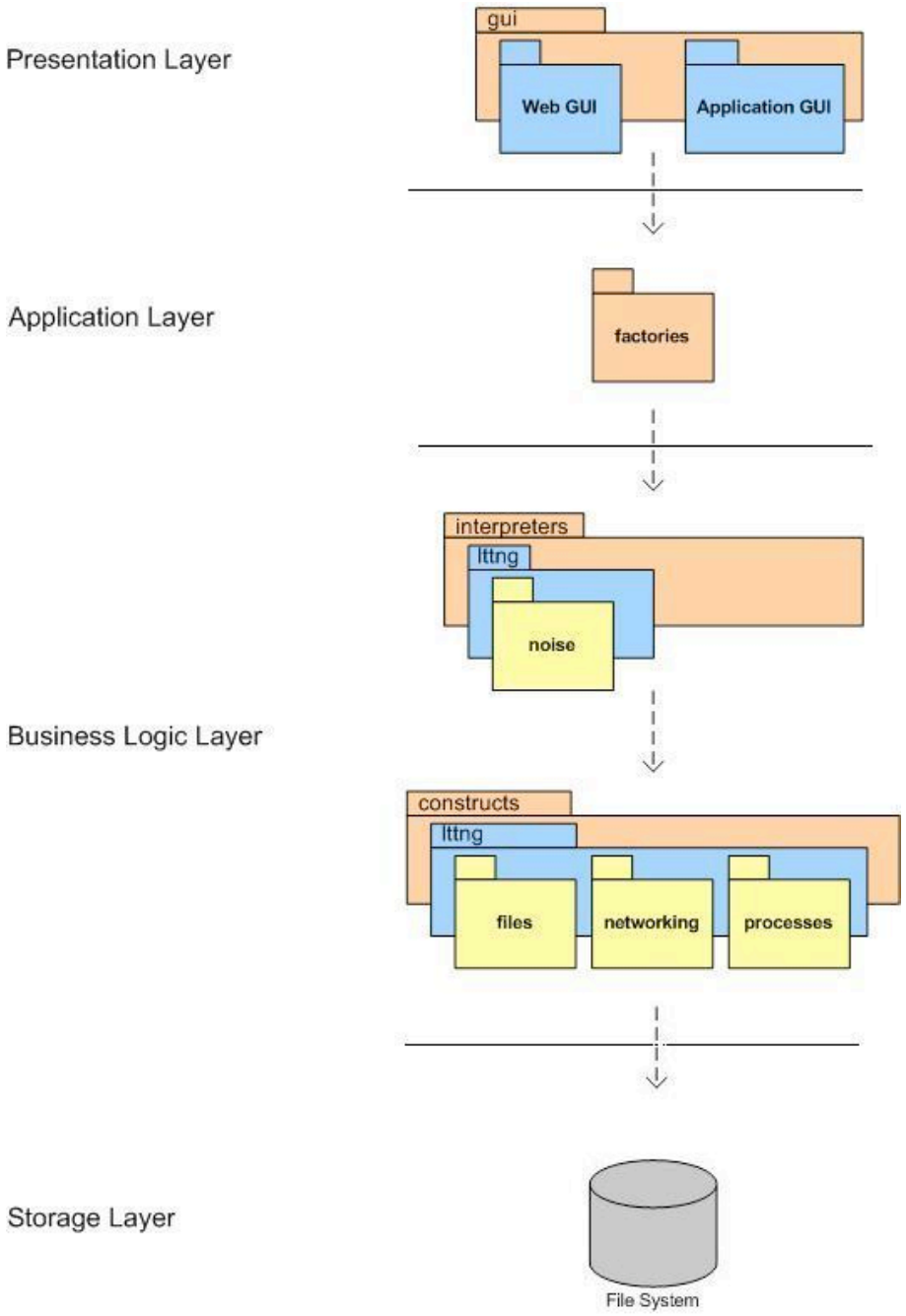

Figure 52. Vertical Partitioning

The UML class diagram [41] of our data model was designed following a number of design patterns [48] such as: Composite pattern [45, 48], Strategy pattern [45, 48], Factory Method Pattern [45, 48] and Façade pattern [45, 48], as shown in Figure 53.

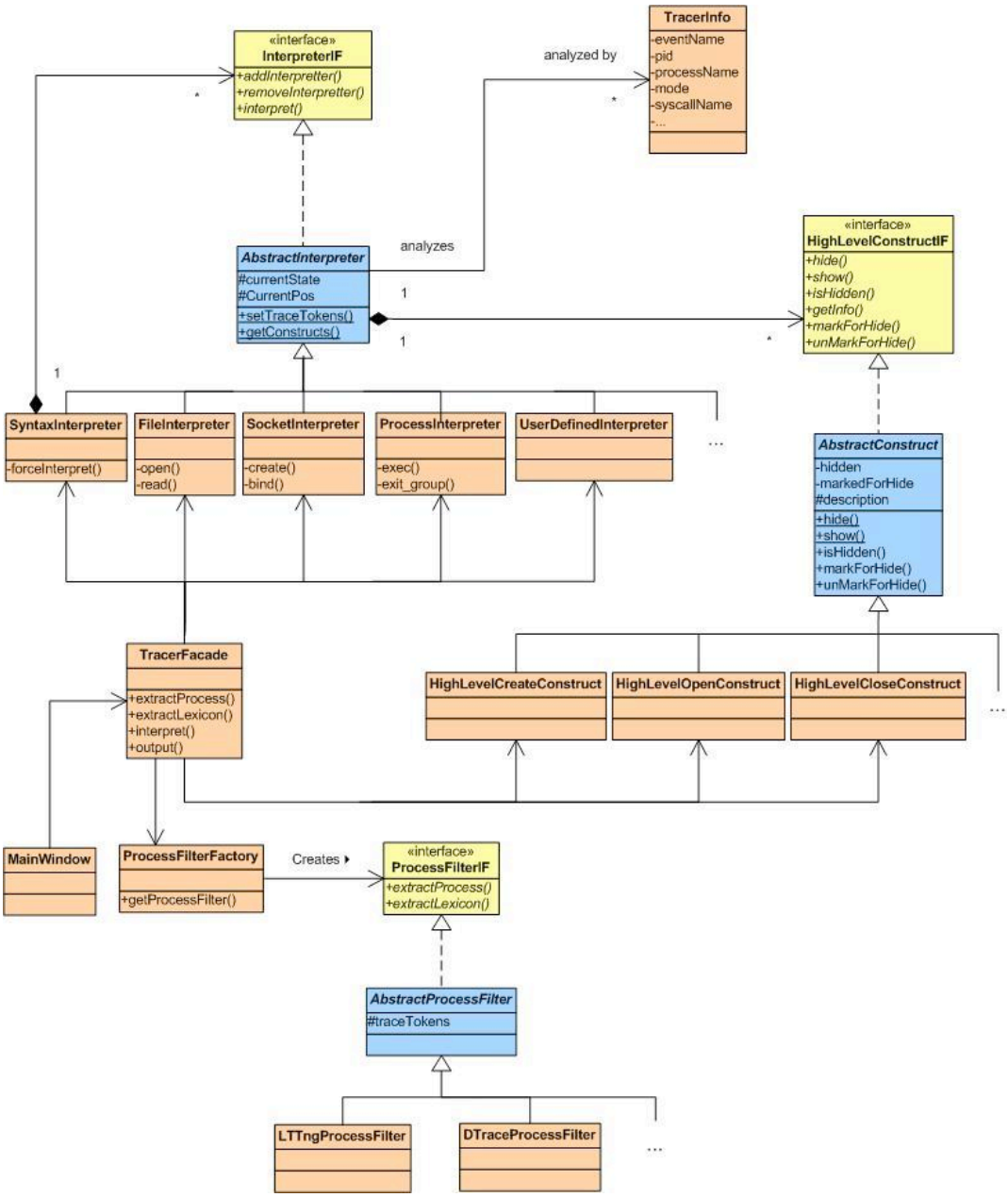

Figure 53. Class Diagram

To explain the class diagram we will focus on the key classes and the application of the design patterns in this diagram. The interpreter design pattern is applied to decouple the construction of the interpreters tree from the client objects. Main components are:

- InterpreterIF: An interface that defines the required functionality for the implementing classes such as: addInterpreter (add a child interpreter to the

tree), removeInterpreter (remove a child interpreter from the tree) and interpret (interpret current token). This interface is equivalent to the ComponentIF in the composite pattern.

- AbstractInterpreter: An abstract class that defines common behaviour and data member for the implementing classes.
- SyntaxInterpreter: This is the composite class that is configured with other interpreters and encapsulates the interpretation logic by insuring that the current trace token (event) is properly interpreted by one of the interpreters.
- UserDefinedInterpreter: A leaf class that defines the required logic to interpret tokens by using XML patterns.
- FileInterpreter, SocketInterpreter, ProcessInterpreter, etc.: Leaf classes that provide the required logic to interpret the corresponding types of tokens (events).

The strategy pattern is applied in two places to provide common interface for both the high-level constructs that appear in the abstracted trace, and the filters that are used to extract events from the original trace.

Main components for the high-level constructs:

- HighLevelConstructIF: An interface that defines the required functionality for implementing classes such as: hide (hide the high-level construct) and show (show the high-level construct). This interface is equivalent to the StrategyIF in the strategy pattern.
- AbstractConstruct: An abstract class that defines common behaviour and data members for the implementing classes.

- HighLevelCreate, HighLevelOpen, HighLevelClose, etc.: Concrete strategy classes that provide the required logic to get the high-level construct information.

Main components for the filters:

- ProcessFilterIF: An interface that defines the required functionality for the implementing classes such as: extractProcess (extract the events for a given process from the original trace) and extracLexicon (tokenize the trace). This interface is equivalent to the StrategyIF in the strategy pattern.
- AbstractProcessFilter: An abstract class that defines common behaviour and data members for the implementing classes.
- LTTngProcessFilter, DtraceProcessFilter, etc.: Concrete strategy classes that provide the required logic to filter the original trace according to its given type.

Factory method pattern is used to decouple the creation of process filter classes from the requesting classes, and it is represented by the class ProcessFilterFactory. The Façade pattern is applied to simplify access to the subsystems through the GUI objects by hiding the communications between those subsystems, and it is presented by the class TracerFacade.

This design leads to a number of advantages that could be summarized in what follows:

- Adding patterns for traces generated from new tools can easily be done by defining appropriate XML files and sub-classing AbstractProcessFilter class.
- Adding new patterns can be easily done by sub-classing AbstractInterpreter class.

- Adding new constructs can easily be done by sub-classing AbstractHighLevelConstruct class.
- Multiple implementations representing different trace formats can be applied using the same interfaces.
- Noise interpreters are marked with the NoiseIF interface which makes it easy to control the noise patterns.
- High-level constructs can be further abstracted (e.g. many open and read file operations can be represented at a high-level as file operations).
- High-level constructs can be hidden or shown by marking/unmarking them for hide.

It should also be mentioned that the tool was designed to accept patterns defined as external XML files [49], which makes it even easier to add new patterns by just defining them in a specific format that corresponds to a predefined XML pattern template file. Also, the XML pattern library provides a total separation between the patterns and the programming language used to abstract the trace, which makes it easy to adopt those patterns to new technologies, frameworks, or languages. However, using XML files affects the performance of the algorithm due to the increase in the number of accesses to the hard disk. This drawback was handled by proposing an optimized algorithm for accessing the files.

The following pseudo code summarizes the optimized algorithm:

```
GET next event from the tokens
FOR every pattern in stack DO
   IF pattern can interpret token THEN
              IF pattern is in final state THEN
                    Generate HIGH LEVEL CONSTRUCT
```

```
                        PULL pattern from stack
                    END IF
                    break;
            END IF
        END FOR
        IF token not interpreted THEN
                FOR every pattern in the XML library DO
                    PARSE pattern
                    IF pattern can interpret token THEN
                                PUSH pattern file into stack
                                IF pattern is in final state THEN
                                        Generate HIGH LEVEL CONSTRUCT
                                        PULL pattern file from stack
                                END IF
                                break;
                    END IF
                END FOR
        END IF
```

We explain the algorithm by giving the following example, while omitting the XML format for simplicity:

Suppose that the pattern library consists of five patterns LIB={p1, p2, ..., p5}, where each of these patterns consists of two events, as follows:

p1 = [p1_entry, p1_exit], p2 = [p2_entry, p2_exit], ..., p5 = [p5_entry, p5_exit].

Let T be a trace which has the following events:

T = [p2_entry, p2_exit, p4_entry, p4_exit]

Starting from the first event p2_entry, the stack is empty; the algorithm will search the patterns by opening their files, reading the content and matching it to the current event.

When there is a match, that is, the algorithm has reached the pattern p2, the matching pattern is pushed into the stack (S = {p2})

After that, the second event (p2_exit) is read, the stack is not empty; so, instead of reading the patterns from the files, the algorithm would search the stack for a pattern with the expected event matching the current event. If this is the case then the algorithm has reached the pattern p2, that pattern is pulled out of the stack, because p2_exit is the last event, and a high level event is generated. Then, the third event (p4_entry) is read, the stack is empty; the algorithm will search the patterns by opening their files, reading the content and matching it to the current event. When there is a match, that is, the algorithm has reached the pattern p4, the matching pattern is pushed into the stack (S = {p4}). Finally, the fourth event (p4_exit) is read, the stack is not empty; so, instead of reading the patterns from the files, the algorithm would search the stack for a pattern with the expected event matching the current event. The algorithm has reached the pattern p4, the pattern is pulled out of the stack, because p4_exit is the last event, and a high level event is generated.

So, the basic idea is that when we have a trace event that matches a pattern event, it is better we keep the pattern in memory, as the next trace event has the chance to match the next pattern event. This would reduce the access to the hard disk and improve the performance of the algorithm.

The XML pattern file has the following format, which is subject to further modifications and improvements.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<pattern name="Sample Pattern" type="HighLevelSampleConstrcut" noise="false">

  <event name="syscall_entry" syscall_name="sys_sample;sys_sample1" order="1"
prev_state="IGNORE" current_state="SYSCALL_SAMPLE">

  </event>

  <event name="sample" order="2" prev_state="SYSCALL_SAMPLE"
current_state="SYSCALL_SAMPLED">

  </event>

  <event name="syscall_exit" order="LAST" prev_state="SYSCALL_SAMPLED"
current_state="USER_MODE_SAMPLED">

  </event>

</pattern>
```

The following tables explain each tag and its attributes.

Table 4. Pattern element attributes

Element	Type	Description	Mandatory
pattern	Tag element	XML root element that surrounds all other elements	YES
name	Attribute	The name that will appear in the high level construct	YES
type	Attribute	The corresponding high level object	NO
noise	Attribute	Indicates whether to display this construct in the output or not	YES

Table 5. Event element attributes

Element	Type	Description	Mandatory
event	Tag element	XML element that defines an event that corresponds to a trace event	YES
name	Attribute	The name of the trace event	YES
order	Attribute	The order of the event in its context. When set to LAST it means that it is the last event in this pattern	YES
prev_state	Attribute	The state of the pattern before hitting this event	YES
current_state	Attribute	The new state of the pattern after hitting this event	YES
syscall_name	Attribute	The name of the system call. It is only provided when the current event is a system call	NO

Using the flexible XML files, we were able to add a number of patterns, for test purposes, by creating new XML files while the application was running, and then

executing the abstraction process to get abstracted traces that included the newly added patterns.

The following example shows how to define the pattern Duplicate File Descriptor which corresponds to the execution of sys_dup system call using the XML pattern template that we have developed.

```
<?xml version="1.0" encoding="UTF-8"?>

<pattern name="Duplicate File Descriptor" type="HighLevelDupConstrcut" noise="false">

  <event name="syscall_entry" syscall_name="sys_dup" order="1" prev_state="IGNORE"
current_state="SYSCALL_DUP">

  </event>

  <event name="syscall_exit" order="LAST" prev_state="SYSCALL_DUP"
current_state="USER_MODE_DUP">

  </event>

</pattern>
```

From the example we can see that the pattern name is set to Duplicate File Descriptor, and this is the name that will appear in the resulting high level construct, while this pattern is not indicated as a noise pattern and will be displayed in the output.

Also, we can see that the first expected event (order is set to 1) in this pattern is a syscall_entry event, with the system call name being sys_dup, and the current state SYSCALL_DUP. The last event (order is set to LAST) in this pattern is syscall_exit, where the current state is set to USER_MODE_DUP. In summary, it is possible to say that we have a pattern that consists of two events that appear in a predefined order, and that whenever this pattern appears in the original trace it will be replaced

with one event in the abstracted trace with its name being Duplicate File Descriptor while preserving other information such as the event name and the system call name.

Using XML files, it was also possible to test the tool against traces generated from DTrace [29] running under FreeBSD operating system [30]. However, in this thesis the main focus is on traces generated from the LTTng tracer running under the Ubuntu operating system [28].

The tool was implemented using Java Standard Edition Development Kit version 6 [36], while the XML files where parsed using stax (Stream API for XML) library [31, 50], where the XML file is read, and the events are provided iteratively through an iterator interface.

4.2. Target Systems

We applied our approach to five large traces which were generated while running five different processes. Each of these applications was executed while LTTng tracer was running and generating the kernel traces. After that, we extracted the execution traces corresponding to the application process and applied our algorithm and displayed the results.

One process was the java virtual machine (JVM) [36] which was running a distributed file server and a client, where the client requests a certain file from the server, and then operates on that file, that is, reads, modifies and saves changes. Another process was the Eclipse framework [37] through which a new project was created and a number of classes were created, executed and deleted. The third process was Gedit [38], which was used to edit files, that is, create, read, and save. The fourth process

was the GIMP image editor[1], which was used to edit (create, modify, and save) images. The final process was Firefox web browser[2], which was used to surf the web by connecting to certain web servers to fetch the requested pages through HTTP requests which are built over TCP sockets. The description of the target systems and the scenarios used to generate the traces are provided in Table 6.

Table 6. **Target systems' main scenarios**

Target System	Scenario
JVM	• Networking • Read • Write • Execute
Eclipse	• Create new files • Read • Write • Execute
Gedit	• Create new files • Read • Write
GIMP	• Create new files • Read • Write
Firefox	• Networking

4.3. Generating Traces

As mentioned in the previous section, each application was executed while running LTTng. The resulting trace is usually large, as LTTng is actually tracing the Linux kernel and not a certain process. However, we were still able to extract traces related to our processes by only copying the LTTng event lines that have the same process name as the traced processes.

[1] http://www.gimp.org/
[2] http://www.mozilla.com/en-US/firefox/personal.html

For each application we provide a sample trace that shows how low-level events are structured within the trace following certain patterns.

A sample trace for JVM application:

```
kernel.syscall_entry: 442192.652062922 (/tmp/trace12/kernel_1), 23566, 23566, /usr/bin/java, , 23565, 0x0, SYSCALL { ip = 0xb7f78430, syscall_id = 102 [sys_socketcall+0x0/0x300] }

net.socket_call: 442192.652063029 (/tmp/trace12/net_1), 23566, 23566, /usr/bin/java, , 23565, 0x0, SYSCALL { call = 1, a0 = 1 }

net.socket_create: 442192.652063137 (/tmp/trace12/net_1), 23566, 23566, /usr/bin/java, , 23565, 0x0, SYSCALL { family = 1, type = 1, protocol = 0, sock = 0xd563d340, ret = 3 }

kernel.syscall_exit: 442192.652063244 (/tmp/trace12/kernel_1), 23566, 23566,

/usr/bin/java, , 23565, 0x0, USER_MODE { ret = 3 }
```

A sample trace for Eclipse application:

```
kernel.syscall_entry:160027.406525216 (/tmp/trace10/kernel_1), 14773, 14661, /usr/lib/eclipse/eclipse, , 14771, 0x0, SYSCALL { ip = 0xb776e430, syscall_id = 5 [sys_open+0x0/0x40] }

fs.open: 160027.406542949 (/tmp/trace10/fs_1), 14773, 14661, /usr/lib/eclipse/eclipse, , 14771, 0x0, SYSCALL { fd = 108, filename = "/home/administrator/workspace/GUI/src/MainWindow.java" }

kernel.syscall_exit: 160027.406544404 (/tmp/trace10/kernel_1), 14773, 14661, /usr/lib/eclipse/eclipse, , 14771, 0x0, USER_MODE { ret = 108 }
```

A sample trace for Gedit application:

```
kernel.syscall_entry:162024.528807231 (/tmp/trace12/kernel_0), 15286, 15286, /usr/bin/gedit, , 15270, 0x0, SYSCALL { ip = 0xb7885430, syscall_id = 120 [ptregs_clone+0x0/0x40] }

kernel.sched_migrate_task: 162024.528817193 (/tmp/trace12/kernel_0), 15286, 15286, /usr/bin/gedit, , 15270, 0x0, SYSCALL { pid = 15286, state = 256, dest_cpu = 0 }

kernel.process_fork: 162024.529118165 (/tmp/trace12/kernel_0), 15286, 15286, /usr/bin/gedit, , 15270, 0x0, SYSCALL { parent_pid = 15286, child_pid = 15287, child_tgid = 15287 }
```

```
kernel.sched_migrate_task:162024.529120286
(/tmp/trace12/kernel_0),  15286,  15286,  /usr/bin/gedit,  ,
15270, 0x0, SYSCALL { pid = 15287, state = 256, dest_cpu = 1 }

kernel.sched_wakeup_new_task:162024.529122132
(/tmp/trace12/kernel_0),  15286,  15286,  /usr/bin/gedit,  ,
15270, 0x0, SYSCALL { pid = 15287, state = 0, cpu_id = 1 }

kernel.syscall_exit: 162024.529122929 (/tmp/trace12/kernel_0),
15286, 15286, /usr/bin/gedit, , 15270, 0x0, USER_MODE { ret =
15287 }
```

A sample trace for GIMP application:

```
kernel.syscall_entry:163637.753443775  (/tmp/trace13/kernel_1),
15753,  15753,  /usr/bin/gimp,  ,  15532,  0x0,  SYSCALL  {  ip  =
0xb7742430, syscall_id = 3 [sys_read+0x0/0xb0] }

fs.read: 163637.753447282 (/tmp/trace13/fs_1), 15753, 15753,
/usr/bin/gimp, , 15532, 0x0, SYSCALL { count = 1024, fd = 7 }

kernel.syscall_exit: 163637.753447579 (/tmp/trace13/kernel_1),
15753, 15753, /usr/bin/gimp, , 15532, 0x0, USER_MODE { ret =
32 }

A sample trace for Firefox application:

kernel.syscall_entry: 159008.031429645 (/tmp/trace9/kernel_0),
14531,  14531,  /usr/lib/firefox-3.6.3/firefox-bin,  ,  14527,
0x0,  SYSCALL  {  ip  =  0xb779a430,  syscall_id  =  5
[sys_open+0x0/0x40] }

fs.open: 159008.031436796 (/tmp/trace9/fs_0), 14531, 14531,
/usr/lib/firefox-3.6.3/firefox-bin, , 14527, 0x0, SYSCALL { fd
=  40,  filename  =  "/usr/lib/firefox-addons/searchplugins/en-
US/google.xml" }

kernel.syscall_exit:  159008.031437901  (/tmp/trace9/kernel_0),
14531,  14531,  /usr/lib/firefox-3.6.3/firefox-bin,  ,  14527,
0x0, USER_MODE { ret = 40 }
```

4.4. Applying System Call Abstraction Techniques

After generating the traces and saving them into a text file, we applied the abstraction process to each of them by running our tool which consists of three main parts: The top part, through which the developer can provide the tool with the required information such as the trace source file, the destination file where the results are to

be stored, the process name, whether to show or hide noise (utility) events, and the Patterns Library trace type, as shown in Figure 54.

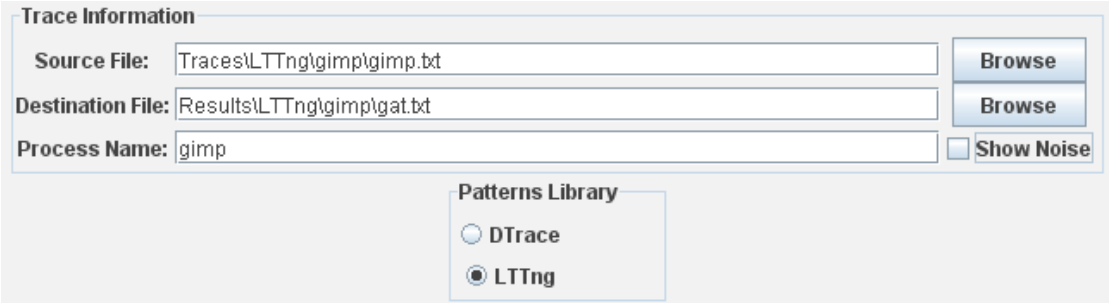

Figure 54. The top part

The middle part, through which the trace resulting from the abstraction process is displayed, with every high-level construct represented in one line, as shown in Figure 55.

SEQ(1) Process Execution: with exec
SEQ(1) File Access: 163597.244837047 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758e41, syscall_id = 33 [sys_access+0x0/0x30] }
SEQ(1) Unknown Event: Syscall name: sys_mmap_pgoff, Params: 163597.244843088 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758f83,
SEQ(1) File Access: 163597.244854104 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758e41, syscall_id = 33 [sys_access+0x0/0x30] }
SEQ(1) File Open: 163597.244869901 (/tmp/trace13/fs_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { fd = 3, filename = "/etc/ld.so.cache" }
SEQ(1) File Stat: 163597.244871232 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758cce, syscall_id = 197 [sys_fstat64+0x0/0x30] }
SEQ(1) Unknown Event: Syscall name: sys_mmap_pgoff, Params: 163597.244872469 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758f83,
SEQ(1) File or Socket Close: 163597.244875229 (/tmp/trace13/fs_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { fd = 3 }
SEQ(1) File Access: 163597.244880180 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758e41, syscall_id = 33 [sys_access+0x0/0x30] }
SEQ(1) File Open: 163597.244902594 (/tmp/trace13/fs_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { fd = 3, filename = "/usr/lib/libgimpwidgets-2.0.so.0" }
SEQ(1) File Read: fd = 3 }
SEQ(1) File Stat: 163597.244909779 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758cce, syscall_id = 197 [sys_fstat64+0x0/0x30] }
SEQ(1) Unknown Event: Syscall name: sys_mmap_pgoff, Params: 163597.244911790 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758f83,
SEQ(1) Unknown Event: Syscall name: sys_mmap_pgoff, Params: 163597.244914121 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758f83,
SEQ(1) Unknown Event: Syscall name: sys_mmap_pgoff, Params: 163597.244924956 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758f83,
SEQ(1) File or Socket Close: 163597.244934321 (/tmp/trace13/fs_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { fd = 3 }
SEQ(1) File Access: 163597.244942562 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758e41, syscall_id = 33 [sys_access+0x0/0x30] }
SEQ(1) File Open: 163597.244951972 (/tmp/trace13/fs_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { fd = 3, filename = "/usr/lib/libgimpmodule-2.0.so.0" }
SEQ(1) File Read: fd = 3 }
SEQ(1) File Stat: 163597.244956021 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758cce, syscall_id = 197 [sys_fstat64+0x0/0x30] }
SEQ(1) Unknown Event: Syscall name: sys_mmap_pgoff, Params: 163597.244957987 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758f83,
SEQ(1) Unknown Event: Syscall name: sys_mmap_pgoff, Params: 163597.244960449 (/tmp/trace13/kernel_0), 15753, 15753, /usr/bin/gimp, , 15532, 0x0, SYSCALL { ip = 0xb7758f83,

Figure 55. The middle part

The bottom part, through which the information related to the original and abstracted traces is displayed, as shown in Figure 56. In this figure, we can see that the number of events in the original trace is 847575 and that after applying our algorithm and removing noise events we obtain an abstracted trace with 243871 events, with a compression ratio equals to (1 – 243871/847575 = 71%)

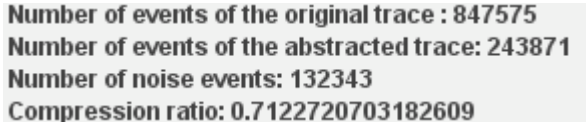

Figure 56. The bottom part

4.5. Results

We discuss the results obtained by applying our abstraction process through quantitative and qualitative analysis. The objective is to answer the following questions:

- How good the compression ratio is?
- How informative the resulting trace is?

4.5.1. Quantitative Analysis

After generating the traces, and abstracting them, we did quantitative analysis by studying the compression ratio to find out how good the algorithm is in terms of compacting the trace. Our initial environment consisted of five traces (resulting from the execution of the five aforementioned processes) with sizes 47271, 186167, 646710, 847575, and 1226985 lines. After abstracting the traces using our tool, we had new traces with the following sizes: 3452, 100523, 309926, 243871, and 465886 lines respectively, which gave us the following compression ratios: 93%, 46%, 52%, 71%, and 62%. Where compression ratio is computed using the following formula: 1 - (abstracted trace size / initial trace size). On the other hand, the number of noise events in each of the traces was: 13444, 10830, 41631, 132343, and 94362 respectively.

When abstracting the trace, we applied noise removal and grouping continuous sequences [19]. Table 7 summarizes the results.

Table 7. Trace abstraction results

Process	Initial Size	Size After Abstraction	Number of Noise Events	Compression Ratio
Eclipse	1226985	465886	94362	62%
GIMP	847575	243871	132343	71%
Firefox	646710	309926	41631	52%
Gedit	186167	100523	10830	46%
JVM	47271	3452	13444	93%

As a result the overall compression ratio was 64.8%, which shows that our approach achieves a good trace compaction ratio.

4.5.2. Qualitative Analysis

To analyze the quality of our work, we compared a sample abstracted trace for a trace generated from a simple C application after removing noise events (Figure 57) with the corresponding application (Figure 58).

From Figure 57 we can see that the first 10 lines include Process Execution, File Access, File Open, File Stat, File Read, and File Close. All these operations are executed to prepare resources for our process, and they do not correspond to any of the source code lines.

```
Abstracted trace after removing noise events:
=============================================
SEQ(1) Process Execution: with exec
SEQ(1) File Access:  442192.435311130 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fc1be1, syscall_id = 33 [sys_access+0x0/0x30] }
SEQ(1) File Access:  442192.435313279 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fc1be1, syscall_id = 33 [sys_access+0x0/0x30] }
SEQ(1) File Open:  442192.435315212 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3, filename = "/etc/ld.so.cache" }
SEQ(1) File Stat:  442192.435315427 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fc1a6e, syscall_id = 197 [sys_fstat64+0x0/0x30] }
SEQ(1) File or Socket Close:  442192.435316609 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3 }
SEQ(1) File Access:  442192.435317791 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fc1be1, syscall_id = 33 [sys_access+0x0/0x30] }
SEQ(1) File Open:  442192.435321551 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3, filename = "/lib/tls/i686/cmov/libc.so.6" }
SEQ(1) File Read:  fd = 3 }
SEQ(1) File Stat:  442192.435323162 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fc1a6e, syscall_id = 197 [sys_fstat64+0x0/0x30] }
SEQ(1) File or Socket Close:  442192.435328963 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3 }
SEQ(1) File Open:  442192.435348299 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3, filename = "output.txt" }
SEQ(1) File Stat:  442192.435349373 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id = 197 [sys_fstat64+0x0/0x30] }
SEQ(1) Process Schedule:  442192.435348192 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { prev_pid = 5457, next_pid = 22438, prev_state = 1 }
SEQ(1) Unknown Event: Event name: add_to_page_cache, Params:  442192.435351200 (/tmp/trace10/mm_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { inode = 11066, sdev = 83
SEQ(1) File Write:  fd = 3 }
SEQ(1) File or Socket Close:  442192.435351629 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3 }
SEQ(1) File Open:  442192.435352596 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3, filename = "output.txt" }
SEQ(1) File Stat:  442192.435353026 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id = 197 [sys_fstat64+0x0/0x30] }
SEQ(1) File Read:  fd = 3 }
SEQ(1) File Stat:  442192.435354315 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id = 197 [sys_fstat64+0x0/0x30] }
SEQ(2) File Write:  fd = 1 }
SEQ(1) File Read:  fd = 3 }
SEQ(1) File or Socket Close:  442192.435357860 (/tmp/trace10/fs_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd = 3 }
SEQ(1) File Write:  fd = 1 }
SEQ(1) Process Exit: Process Exit:  442192.435366024 (/tmp/trace10/kernel_1), 22438, 22438, ./Files, , 29184, 0x0, SYSCALL { pid = 22438 }, Send Signal:  442192.435366669
Results:
========
Number of events of the original trace : 365
Number of events of the abstracted trace : 26
Number of noise events : 123
Compression ratio: 0.9287671232876712
```

Figure 57. Abstracted Trace

```c
#include <stdio.h>

int main(void){
    FILE *fp;
    fp = fopen("output.txt", "w");
    fprintf(fp, "This is a test line\n");
    fprintf(fp, "This is another test line\n");
    fprintf(fp, "This is the last test line");
    fclose(fp);
    fp = fopen("output.txt", "r");
    int c = 0;
    while (c!=EOF) {
        c=fgetc(fp);
            printf("%c", c);
    }
    fclose(fp);
    return 0;
}
```

Figure 58. Corresponding C Application

After that a File Open is executed with the file name parameter set to output.txt and the file descriptor set to 3. If we take a look at our application we can see that this File Open has occurred as a result to the first two lines within the main function. Next, a File Stat is executed to get file information, followed by a File Write that corresponds to fprintf in the source code. File Close is executed in both the source code and the

trace, then the same file is opened again, only that a File Read appears in the trace as a result of executing fgetc, followed by a File Write with a file descriptor set to 1, which means writing to the standard output in correspondence to printf in the source code. Finally, the file is closed and the process is exited.

We also had similar results when analyzing the other applications, where we were able to map the low-level system calls to the events that caused these calls to be invoked. As a result, we can say that the abstracted trace provides a clear view of the process execution that reflects the source code of the process in case it was available.

Chapter 5. Conclusions

In this thesis, we introduced techniques to abstract execution traces resulting from the Linux kernel. Our main focus was on the idea of building a library of system calls patterns, and using it to abstract low-level traces. We also provided a number of case studies, and assessed our work in terms of quantity where we had an average compression ratio of 64.8%, and in terms of quality where we had smaller abstracted traces with lines corresponding to the execution of a process.

The following sections summarize our research contributions and the possible future directions.

5.1. Research Contributions

In this research we defined system call traces and introduced techniques to abstract them. The main contributions of this research can be summarized through the following points:

Our approach is based on building a pattern library that consists of patterns of the most common operations in Linux. This library was built by studying the Linux kernel system call mechanism and by generating system call traces using LTTng tracer, and comparing these traces to the list of processes that were run to perform specific tasks. We also defined noise (utility) patterns that result from memory management operations and page faults.

Then, we introduced an algorithm to abstract the system call traces by using the pattern library to detect different patterns in the trace and replace them with higher level constructs that are easier to understand, while hiding utility patterns.

We have also developed a tool that support the trace abstraction process presented in this thesis. Finally, we applied our techniques to traces generated from four different processes that were run under the Linux operating system. We were able to reach a 65% compression ratio while preserving the internal state of the processes and showing the main operations as high-level constructs that are easy to understand.

5.2. Future Directions

One direction for future work would be to study more patterns to fully document the system calls patterns. Once completed, we can use this library to abstract any trace generated from the Linux kernel.

Another direction would be to use the abstracted traces to monitor the system for the purpose of detecting any malicious behaviour, bug, or performance bottlenecks. Also, abstracted traces could be used to compare traces resulting from identical or different scenarios, for any of the reasons mentioned above.

A third direction would be to abstract the resulting traces into a higher-level of abstraction by following the same techniques, that is, implementing the higher-level constructs, over the already defined constructs, as part of the class hierarchy, or defining them in new XML files.

A fourth direction would be to detect non-contiguous repeated patterns in the abstracted traces and find ways of representing them only once in the final trace.

Finally, it would be interesting to define a formal language that describes the traces, and could be further used to formally describe scenarios and to parse traces according to these scenarios.

Bibliography

[1] Andrew Chan, Reid Holmes, Gail C. Murphy and Annie T.T. Ying. Scaling an Object-oriented System Execution Visualizer through Sampling. In Proc. of the 11th IEEE International Workshop on Program Comprehension (IWPC'03), pages 237-244, 2003

[2] Adrian Kuhn and Orla Greevy. Exploiting the Analogy between Traces and Signal Processing. In Proc. of IEEE International Conference on Software Maintenance (ICSM 2006), pages 320-329. IEEE Computer Society Press: Los Alamitos CA, 2006

[3] Wim De Pauw, David Lorenz, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides and Jeaha Yang. Visualizing the Execution of Java Programs. In Proc. of the International Seminar on Software Visualization, pages 151-162, Dagstuhl Castle, Wadern, 2002

[4] James R. Larus. Whole Program Paths. In Proc. of the SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI 99), May 1999, Atlanta Georgia

[5] Dean Jerding and Spencer Rugaber. Using Visualization for Architectural Localization and Extraction. In Proc. of the 4th Working Conference on Reverse Engineering, October 1997, the Netherlands, IEEE Computer Society, pp. 56-65

[6] Tarja Systä. Understanding the Behavior of Java Programs. In Proc. of the 7th Working Conference on Reverse Engineering, Australia, Brisbane, 2000, pp. 214-223.

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, Linux Device Drivers, Third Edition, O'Reilly, February 2005

[8] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, 2nd Edition, O'Reilly, December 2002

[9] Jialong He . LINUX System Call Quick Reference, URL: http://www.digilife.be/quickreferences/QRC/LINUX%20System%20Call%20Quick%20Reference.pdf. Last Accessed: September 2010

[10] Mathieu Desnoyers and Michel R. Dagenais. Tracing for Hardware, Driver, and Binary Reverse Engineering in Linux. Code Breakers Journal Vol. 1, No. 2, 2006

[11] Mathieu Desnoyers. Linux Trace Toolkit Viewer User Guide. URL: http://lttng.org/files/lttv-doc/user_guide/c42.html. Last Accessed: September 2010

[12] Mathieu Desnoyers and Michel R. Dagenais. The LTTng tracer : A Low Impact Performance and Behavior Monitor for GNU/Linux. In Proc. of Ottawa Linux Symposium 2006.

[13] Hassan Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, July, 2000

[14] Linux Init Process. URL:

http://www.yolinux.com/TUTORIALS/LinuxTutorialInitProcess.html.

Last Accessed: June 2010

[15] Sockets Tutorial. URL: http://www.linuxhowtos.org/C_C++/socket.htm. Last Accessed: June 2010

[16] A. Hamou-Lhadj and Timothy Lethbridge. Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools. In Proc. of the 10th International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, pages 559–568, 2005

[17] A. Hamou-Lhadj and Timothy Lethbridge. Compression Techniques to Simplify the Analysis of Large Execution Traces. In Proc. of the 10th International Workshop on Program Comprehension (IWPC), pages 159-168, Paris, France, 2002

[18] A. Hamou-Lhadj and Timothy Lethbridge. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In Proc. 14th Int. Conf. on Program Comprehension (ICPC), pages 181–190. IEEE, 2006

[19] A. Hamou-Lhadj. Techniques to Simplify the Analysis of Execution Traces for Program Comprehension. PhD Thesis, Ottawa-Carleton Institute for Computer Science, School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario, Canada, 2005

[20] A. Hamou-Lhadj and Timothy Lethbridge. Reasoning about the Concept of Utilities. ECOOP PPPL, Oslo, Norway, June 14, 2004

[21] Valiente G. Simple and Efficient Tree Pattern Matching. Research Report E-08034, Technical University of Catalonia, 2000

[22] Bas Cornelissen and Leon Moonen. On Large Execution Traces and Trace Abstraction Techniques. Report TUD-SERG-2008-005. Delft University of Technology, Software Engineering Research Group, Technical Report Series

[23] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing test suites to aid in software understanding. In Proc. of the 11th European Conf. on Software Maintenance and Reengineering (CSMR), pages 213–222. IEEE, 2007

[24] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In Proc. of the Data Compression Conference (DCC '97). Snowbird, UT: IEEE Computer Society, 1997.

[25] C. Bennett, D. Myers, M. A. Storey, D.M. German, D. Ouellet, M. Salois, and P. Charland. A Survey and Evaluation of Tool Features for Understanding Reverse Engineered Sequence Diagrams. Journal of Software Maintenance and Evolution: Research and Practice, March 2008

[26] Syscalls(2) – Linux Man Page. URL: http://linux.die.net/man/2/syscalls. Last Accessed October 2010

[27] Binh Nguyen, Linux File System Hierarchy, 2003, http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/Linux-Filesystem-Hierarchy.pdf

[28] Andrew Hudson, Paul Hudson, Ubuntu Unleashed, Sams, August 2006

[29] DTrace. URL: http://www.oracle.com/technetwork/systems/dtrace/dtrace/index-jsp-137532.html. Last Accessed: October 2010

[30] Greg Lehey, The Complete FreeBSD: Documentation from the Source, 4th edition, O'Reilly, April 2005

[31] Brett McLaughlin, Justin Edelson, Java and XML, Third Edition, O'Reilly, December 2006

[32] Frank Ch. Eigler, Systemtap tutorial, March 24, 2010

http://sourceware.org/systemtap/tutorial.pdf

[33] WindRiver Workbench. URL:
http://lttng.org/tracingwiki/index.php/WindRiver_Workbench.
Last Accessed: June 2010

[34] Zealcore System Debugger. URL:
http://lttng.org/tracingwiki/index.php/Zealcore_System_Debugger
Last Accessed May 2010

[35] http://lttng.org/tracingwiki/index.php/File:Zealcore_img.png

[36] Java SE Downloads. URL:
http://www.oracle.com/technetwork/java/javase/downloads/index.html. Last Accessed: October 2010

[37] Eclipse. URL: http://www.eclipse.org/. Last Accessed: September 2010

[38] Gedit Text Editr. URL: http://projects.gnome.org/gedit/. Last Accessed: October 2010

[39] Abdelwahab Hamou-Lhadj, Timothy C. Lethbrridge, A Survey of Trace Exploration Tools and Techniques. In Proc. 2004 Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON' 04), pages 42–55, 2004.

[40] http://lttng.org/tracingwiki/index.php/File:Windriver_workbench_img.png

[41] Kim Hamilton, Russell Miles, Learning UML 2.0, O'Reilly, April 2006

[42] Linux Man Page. URL: http://linux.die.net/man/2/set_thread_area. Last Accessed: July 2010

[43] What is RCU, Fundamentally?. URL: http://lwn.net/Articles/262464/. Last Accessed: May 2010

[44] LTTng Trace Format. URL: http://benno.id.au/docs/lttng_data_format.pml. Last Accessed: July 2010

[45] Mark Grand, Patterns in Java, Volume 1—A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition, Wiley Publishing, Inc., 2002

[46] http://lttng.org/tracingwiki/index.php/File:VTune_img.png

[47] Wolfgang Mauerer, Professional Linux® Kernel Architecture, Wiley Publishing, Inc., 2008

[48] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994

[49] Mitch Amiano, Conrad D'Cruz, Kay Ethier, and Michael D. Thomas, XML Problem – design – Solution, Wiley Publishing, Inc., 2006

[50] The Java EE 5Tutorial For Sun Java System Application Server 9.1, Oracle, 2010